

This chapter provides complete documentation for the Object Collection Service specification.

Contents

This chapter contains the following sections.

Section Title	Page
“Overview”	17-2
“Service Structure”	17-2
“Combined Collections”	17-10
“Restricted Access Collections”	17-14
“The CosCollection Module”	17-15
Appendix A, “OMG Object Query Service”	17-124
Appendix B, “Relationship to Other Relevant Standards”	17-133
Appendix C, “References”	17-138

17.1 Overview

Collections support the grouping of objects and support operations for the manipulation of the objects as a group. Common collection types are queues, sets, bags, maps, etc. Collection types differ in the “nature of grouping” exposed to the user. “Nature of grouping” is reflected in the operations supported for the manipulation of objects as members of a group. Collections, for example, can be ordered and thus support access to an element at position “i” while other collections may support associative access to elements via a key. Collections may guarantee the uniqueness of elements while others allow multiple occurrences of elements. A user chooses a collection type that matches the application requirements based on manipulation capabilities.

Collections are foundation classes used in a broad range of applications; therefore, they have to meet the general requirement to be able to collect elements of arbitrary type. On the other hand, a collection instance usually is a homogenous collection in the sense that all elements collected are of the same type, or support the same single interface.

Sometimes you may not want to do something to all elements in a collection, but only treat an individual object or traverse a collection explicitly (not implicitly via a collection operation). To enable this, a pointer abstraction often called an iterator is supported with collections. For example, an iterator points to an element in a collection and processes the element pointed to. Iterators can be moved and used to visit elements of a collection in an application defined manner. There can be many iterators pointing to elements of the same collection instance.

Normally, when operating on all elements of a collection, you want to pass user-defined information to the collection implementation about what to do with the individual elements or which elements are to be processed. To enable this, function interfaces are used. A collection implementation can rely on and use the defined function interface. A user has to specialize and implement these interfaces to pass the user-defined information to the implementation. A function interface can be used to pass element type specific information such as how to compare elements or pass a “program” to be applied to all elements.

17.2 Service Structure

The purpose of an Object Collection Service is to provide a uniform way to create and manipulate the most common collections generically. The Object Service defines three categories of interfaces to serve this purpose.

1. **Collection interfaces and collection factories.** A client chooses a collection interface which offers grouping properties that match the client’s needs. A client creates a collection instance of the chosen interface using a collection factory. When creating a collection, a client has to pass element type specific information such as how to compare elements, how to test element equality, or the type checking desired. A client uses collections to manipulate elements as a group. When

a collection is no longer used it may be destroyed - this includes removing the elements collected, destroying element type specific information passed, and the iterators pointing to this collection.

2. **Iterator interfaces.** A client creates an iterator using the collection for which it is created as factory. A client uses an iterator to traverse the collection in an application defined manner, process elements pointed to, mark ranges, etc. When a client no longer uses an iterator, it destroys the iterator.
3. **Function interfaces.** A client creates user-defined specializations of these interfaces using user-defined factories. Instances are passed to a collection implementation when the collection is created (element type specific information) or as a parameter of an operation (for example, code to be executed for each element of the collection). Instances of function interfaces are used by a collection implementation rather than by a client.

17.2.1 Combined Property Collections

The Object Collection Service (or simply Collection Service) defined in this specification aims at being a complete and differentiated offering of interfaces supporting the grouping of objects. It enables a user to make a choice when following the rule “pay only for what you use.” With this goal in mind, a very systematic approach was chosen.

Groups, or collections of objects, support operations and exhibit specific behaviors that are mainly related to the nature of the collection rather than the type of objects they collect.

“Nature of the collection” can be expressed in terms of well defined properties.

Ordering of elements

A *previous* or *next* relationship exists between the elements of an *ordered collection* which is exposed in the interface.

Ordering can be sequential or sorted. A sequential ordering can be explicitly manipulated; however, a sorted ordering is to be maintained implicitly based on a sort criteria to be defined and passed to the implementation by the user.

Access by key

A *key collection* allows associative access to elements via a *key*. A key can be computed from an element value via a user-defined key operation. Furthermore, key collections require key equality to be defined.

Element equality

An *equality collection* exploits the property that a test for element equality is defined (i.e., it can be tested whether an element is equal to another in terms of a user-defined element equality operation). This enables a test on containment, for example.

Uniqueness of entries

A collection with *unique* entries allows exactly one occurrence of an element key value, not *multiple* occurrences.

Meaningful combinations of these basic properties define “collections of differing nature of grouping.” Table 17-1 provides an overview of meaningful combinations. The listed combinations are described in more detail in the following section.

Table 17-1 Interfaces derived from combinations of collection properties

		Unordered		Ordered		
				Sorted		Sequen- tial
		Unique	Multiple	Unique	Multiple	Multiple
Key (Key equality must be specified)	Element Equality	Map	Relation	Sorted Map	Sorted Relation	
	No Element Equality	KeySet	KeyBag	Key Sorted Set	Key SortedBag	
No Key	Element Equality	Set	Bag	SortedSet	Sorted Bag	Equality Sequence
	No Element Equality		Heap			Sequence

Properties are mapped to interfaces - each interface assembling operations that exploit these properties. These interfaces are combined via multiple inheritance and form an *abstract interface hierarchy*. Abstract means that no instance of such a class can be instantiated, an attempt to do so may raise an exception at run-time. Leaves of this hierarchy represent concrete interfaces listed in the table above and can be instantiated by a user. They form a complete and differentiated offering of collection interfaces.

Restricted Access Collections

Common data structures based on these properties sometimes restrict access such as queues, stacks, or priority queues. They can be considered as restricted access variants of **Sequence** or **KeySortedBag**. These interfaces form their own hierarchy of *restricted access interfaces*. They are not incorporated into the hierarchy of combined properties because a user of restricted access interfaces should not be bothered with inherited operations which cannot be used in these interfaces. Nevertheless, to support several “views” on an interface, a restricted users view of a queue and an unrestricted system administrators view to the same queue instance, the restricted access collections are defined in a way that allows combining them with the combined properties collections via multiple inheritance.

All collections are unbounded (there is no explicit bound set) and controlled by the collections; however, it depends on the quality of service delivered whether there are “natural” limits such as the size of the paging space.

Collection Factories

For each concrete collection interface specified in this specification there is one corresponding collection factory defined. Each such factory offers a typed create operation for the creation of collection instances supporting the respective collection interface.

Additionally, a generic extensible factory is specified to enable the usage of many implementation variants for the same collection interface. This extensible generic factory allows the registration of implementation variants and their user-controlled selection at collection creation time.

Information to be passed to a collection at creation time is the element and key type specific information that a collection implementation relies on. That is, one passes the information how to compare element keys, how to test equality of element keys, type checking relevant information, etc. Which type of information needs to be passed depends on the respective collection interface.

17.2.2 Iterators

Iterators, as defined in this specification, are more than just simple “pointing devices.”

Iterator hierarchy

The service defines a hierarchy of iterators which parallels the collection hierarchy.

The top level iterator is generic in the sense that it allows iteration over all collections, independent of the collection type because it is supported by all collection types. The ordered iterator adds some capabilities useful for all kinds of ordered collections. Iterators further down in the hierarchy add operations exploiting the capabilities of the corresponding collection type. Not. Each iterator type is supported by each collection type. For example, a `KeyIterator` is supported only by collection interfaces derived from `KeyCollection`.

Iterators are tightly intertwined with collections. An iterator cannot exist independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this, and only this, collection.

Generic and iterator centric programming

Iterators on the one hand are pointer abstractions in the sense of simple pointing devices. They offer the basic capabilities you can expect from a pointer abstraction. One can reset an iterator to a start position for iteration and move or position it in different ways depending on the iterator type.

There are essentially two reasons to embellish an iterator with more capabilities.

1. To support the processing of very large collections to allow for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in “fine grains” via an iterator passed to a client.
2. To enrich the iterator with more capabilities is to strengthen the support for the generic programming model as introduced with ANSI STL to the C++ world.

One can retrieve, replace, remove, and add elements via an iterator. One can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore, an iterator can have a `const` designation which is set when created. A `const` iterator can be used for access only.

The `reverse` iterator semantics is supported. No extra interfaces are specified to support this but a `reverse` designation is set at creation time. An ordered iterator for which the `reverse` designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, combined operations and bulk operations are offered.

- Combined operations are combinations of simple iterator operations often used in loops.
- Bulk operations support retrieving, replacing, and adding many elements within one operation.

Managed Iterators

All iterators are managed in the sense that iterators never become undefined; therefore, they do not lead to undefined behavior. Common behavior of iterators in class libraries today is that iterators become undefined when the collection content is changed. For example, if an element is added the side effect on iterators of the collection is unknown. Iterators do not “know” whether they are still pointing to the same element as before, still pointing to an element at all, or pointing “outside” the collection. One cannot even test the state. This is considered unacceptable behavior in a distributed environment.

The iterator model used in this specification is a managed iterator. Managed iterators are “robust” to modifications of the collection. A managed iterator is always in one of the following defined testable states:

- *valid* (pointing to an element of the collection)
- *invalid* (pointing to nothing; comparable to a NULL pointer)
- *in-between* (not pointing to an element, but still “remembering” enough state to be valid for most operations on it).

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum

within the collection. There is nothing the managed iterator can point to. Nevertheless, managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a `set_to_next_element()` for example) without resetting it first. For more information, see “The Managed Iterator Model” on page 17-85.

17.2.3 Function Interfaces

The Object Collection service specifies function interfaces used to pass user-defined information to the collection implementation (either at creation time or as parameters of operations). The most important is the **Operations** interface discussed in more detail below.

Collectible Elements and Type Safety

Collections are foundation classes used in a broad range of applications. They have to be able to collect elements of arbitrary type and support keys of arbitrary type. Instances of collections are usually homogenous collections in the sense that all elements have the same element type.

Because there is no template support in CORBA IDL today, the requirement “collecting elements of arbitrary type” is met by defining the element type and the key type as a CORBA **any**. In doing so, compile time type checking for element and key type is impossible.

As collections are often used as homogenous collections, dynamic type checking is enabled by passing relevant information to the collection at creation time. This is done by specialization of the function interface **Operations**. This interface defines attributes `element_type` and `key_type` as well as defines operations `check_element_type()` and `check_key_type()` which have to be implemented by the user. Implementations may range from “no type checking at all,” “type code match,” “checking an interface to be supported,” up to “checking constraints in addition to a simple type code checking.” Using the **Operations** interface allows user-defined customization of the dynamic type checking.

Collectible Elements and the Operations Interface

The function interface **Operations** is used to pass a number of other user-defined element type specific information to the collection implementation.

The type checking of relevant information is one sample.

Depending on the properties represented by a collection interface, a respective implementation relies on some element type specific or key type specific information passed to it. For example, one has to pass the information “element comparison” to implement a **SortedSet** or “key equality” to guarantee uniqueness of keys in a **KeySet**. The **Operations** interface is used to pass this information.

The third use of this interface is to pass element or key type specific information that the different categories of implementations rely on. For example, tree-like implementations for a **KeySet** rely on the “key comparison” information and hashing based implementations rely on the information how to hash key values. This information is passed via the **Operations** interface.

A user has to customize the **Operations** interface and to implement the appropriate operations dependent on the collection interface to be used. An instance of the specialized **Operations** interface is passed at collection creation time to the collection implementation.

Collectible Elements of Key Collections

Key collections offer associative access to collection elements via a key. A key is computed from the element value and is user-defined element type specific information to be passed to a collection. The **Operations** interface has an operation **key()** which returns the user-defined key of a given element.

For a specific element type, a user has to implement the element type specific **key()** operation in an interface derived from **Operations**. The key type is a CORBA any. Again this is designed to accommodate generality. Computable keys reflect the data base view on elements of key collections as “keyed elements” where a key is a component of a tuple or is “composed” from several components of a tuple.

17.2.4 List of Interfaces Defined

The Object Collection service offers the following interfaces:

Abstract interfaces representing collection properties and their combinations

- **Collection**
- **OrderedCollection**
- **KeyCollection**
- **EqualityCollection**
- **SortedCollection**
- **SequentialCollection**
- **EqualitySequentialCollection**
- **EqualityKeyCollection**
- **KeySortedCollection**
- **EqualitySortedCollection**
- **EqualityKeySortedCollection**

Concrete collections and their factories

- CollectionFactory, CollectionFactories
- KeySet, KeySetFactory
- KeyBag, KeyBagFactory
- Map, MapFactory
- Relation, RelationFactory
- Set, SetFactory
- Bag, BagFactory
- KeySortedSet, KeySortedSetFactory
- KeySortedBag, KeySortedBagFactory
- SortedMap, SortedMapFactory
- SortedRelation, SortedRelationFactory
- SortedSet, SortedSetFactory
- SortedBag, SortedBagFactory
- Sequence, SequenceFactory
- EqualitySequence, EqualitySequenceFactory
- Heap, HeapFactory

Restricted access collections and their factories

- RestrictedAccessCollection, RACollectionFactory
- Stack, StackFactory
- Queue, QueueFactory
- Deque, DequeFactory
- PriorityQueue, PriorityFactory

Iterator interfaces

- Iterator
- OrderedIterator
- SequentialIterator
- SortedIterator
- KeyIterator
- EqualityIterator
- EqualityKeyIterator

- KeySortedIterator
- EqualitySortedIterator
- EqualitySequentialIterator
- EqualityKeySortedIterator

Function interfaces

- Operations
- Command
- Comparator

17.3 Combined Collections

The overview introduced *properties* and listed the meaningful combinations of these properties that result in consistently defined collection interfaces forming a differentiated offering. In the following sections, the semantics of each combination will be described in more detail and demonstrated by an example.

17.3.1 Combined Collections Usage Samples

Bag, SortedBag

A Bag is an unordered collection of zero or more elements with no key. Multiple elements are supported. As element equality is supported, operations which require the capability “test of element equality” (e.g., test on containment) can be offered.

Example: The implementation of a text file compression algorithm. The algorithm finds the most frequently occurring words in sample files. During compression, the words with a high frequency are replaced by a code (for example, an escape character followed by a one character code). During re-installation of files, codes are replaced by the respective words.

Several types of collections may be used in this context. A Bag can be used during the analysis of the sample text files to collect isolated words. After the analysis phase you may ask for the number of occurrences for each word to construct a structure with the 255 words with the highest word counts. A Bag offers an operation for this, you do not have to “count by hand,” which is less efficient. To find the 255 words with the highest word count, a SortedRelation is the appropriate structure (see “Relation, SortedRelation” on page 17-13). Finally, a Map may be used to maintain a mapping of words to codes and vice versa. (See “Map, SortedMap” on page 17-12).

A SortedBag (as compared to a Bag) exposes and maintains a sorted order of the elements based on a user-defined element comparison. Maintained elements in a sorted order makes sense when printing or displaying the collection content in sorted order.

EqualitySequence

An EqualitySequence is an ordered collection of elements with no key. There is a first and a last element. Each element, except the last one, has a next element and each element, except the first one, has a previous element. As element equality is supported, all operations that rely on the capability “test on element equality” can be offered, for example, locating an element or test for containment.

Example: An application that arranges wagons to a train. The order of the wagons is important. The trailcar has to be the first wagon, the first class wagons are arranged right behind the trailcar, the restaurant has to be arranged right after the first class and before the second class wagons, and so on. To check whether the wagon has the correct capacity, you may want to ask: “How many open-plan carriages are in the train?” or “Is there a bistro in the train already?”

Heap

A Heap is an unordered collection of zero or more elements without a key. Multiple elements are supported. No element equality is supported.

Example: A “trash can” on a desktop which memorizes all objects moved to the trashcan as long as it is not emptied. Whenever you move an object to the trashcan it is added to the heap. Sometimes you move an object accidentally to the trashcan. In that case, you iterate in some order through the trashcan to find the object - not using a test on element equality. When you find it, you remove it from the trashcan. Sometimes you empty the trashcan and remove all objects from the trashcan.

KeyBag, KeySortedBag

A KeyBag is an unordered collection of zero or more elements that have a key. Multiple keys are supported. As no element equality is assumed, operations such as “test on collection equality” or “set theoretical operation” are not offered.

A KeySortedBag is sorted by key. In addition to the operations supported for a KeyBag, all operations related to ordering are offered. For example, operations exploiting the ordering such as “set_to_previous / set_to_next” and “access via position” are supported.

A license server maintaining floating licenses on a network may be implemented using a KeyBag to maintain the licenses in use. The key may be the LicenseId and additional element data may be, for example, the user who requested the license. As usual, more than one floating license is available per product; therefore, many licenses for the same product may be in use. A LicenseId may occur more than once. A user may request a license multiple times, it may also occur that the same LicenseId with the same user occurs multiple times. If a user of the product requests and receives the license, the LicenseId, together with the request data, is added to the licenses in use. If the license is released, it is deleted from the Bag of licenses in use. Sometimes you may want to ask for the number of licenses of a product in use, that is ask for the number of the licenses in use with a given LicenseId.

Access to licenses in use is via the key `LicenseId`. This sample application does not require operations such as testing two collections for equality or set theoretical operations on collections. It is not exploiting element equality; therefore, it can use a `KeyBag` instead of a `Relation` (which would force the user to define element equality).

If you want to list the licenses in use with the users holding the licenses sorted by `LicenseId`, you could make use of a `KeySortedBag` instead of a `KeyBag`.

KeySet, KeySortedSet

A `KeySet` is an unordered collection of zero or more elements that have a key. Keys must be unique. Defined element equality is not assumed; therefore, operations and semantics which require the capability “element equality test” are not offered.

A `KeySortedSet` is sorted by key. In addition to the operations supported for a `KeySet`, all operations related to ordering are offered. For example, operations exploiting the ordering, such as “set_to_previous / set_to_next” and “access via position” are supported.

Example: A program that keeps track of cancelled credit card numbers and the individuals to whom they are issued. Each card number occurs only once and the collection is sorted by card number. When a merchant enters a customer’s card number into the point-of-sales terminal, the collection is checked to determine whether the card number is listed in the collection of cancelled cards. If it is found, the name of the individual is shown and the merchant is given directions for contacting the card company. If the card number is not found, the transaction can proceed because the card is valid. A list of cancelled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Map, SortedMap

A `Map` is an unordered collection of zero or more elements that have a key. Keys must be unique. As defined, element equality is assumed access via the element value and all operations which need to test on element equality, such as a test on containment for an element, test for equality, and set theoretical operations can be offered for maps.

A `SortedMap` is sorted by key. In addition to the operations supported for a `Map`, all operations related to ordering are offered. For example, operations exploiting the ordering like “set_to_previous / set_to_next” and “access via position” are supported.

Example: Maintaining nicknames for your mailing facility. The key is the nickname. Mailing information includes address, first name, last name, etc. Nicknames are unique; therefore, adding a nickname/mailling information entry with a nickname that is already available should fail, if the mailing information to be added is different from the available information. If it is exactly the same information, it should just be ignored. You may define more than one nickname for the same person; therefore, the same element data may be stored with different keys. If you want to update address

information for a given nickname, use the `replace_element_with_key()` operation. To create a new nickname file from two existing files, use a union operation which assumes element equality to be defined.

Relation, SortedRelation

A Relation is an unordered collection of zero or more elements with a key. Multiple keys are supported. As defined element equality is assumed, test for equality of two collections is offered as well as the set theoretical operations.

A SortedRelation is sorted by key. In addition to the operations supported for a Relation, all operations related to ordering are offered. For example, operations that exploit ordering such as “`set_to_previous / set_to_next`” and “access via position” are supported.

A SortedRelation may be used in the text file compression algorithm mentioned previously in the Bag, Sorted Bag example to find the 255 words with the highest frequency. The key is the word count and the additional element data is the word. As words may have equal counts, multiple keys have to be supported. The ordering with respect to the key is used to find the 255 highest keys.

Set, SortedSet

A set is an unordered collection of zero or more elements without a key. Element equality is supported; therefore, operations that require the capability “test on element equality” such as intersection or union can be offered.

A SortedSet is sorted with respect to a user-defined element comparison. In addition to the operations supported for a Set, all operations related to ordering are offered. For example, operations that exploit ordering such as “`set_to_previous / set_to_next`” and “access via position” are supported.

Example: A program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is added to the set. The set does not allow an item to be added if it already is present in the collection; this ensures that a customer does not get two samples of a single product.

Sequence

A Sequence is an ordered collection of elements without a key. There is a first and a last element. Each element (except the last one) has a next element and each element (except the first one) has a previous element. No element equality is supported; therefore, multiples may occur and access to elements via the element value is not possible. Access to elements is possible via position/index.

Example: A music editor. The Sequence is used to maintain tokens representing the recognized notes. The order of the notes is obviously important for further processing of the melody. A note may occur more than once. During editing, notes are accessed by position and are removed, added, or replaced at a given position. To print the result, you may iterate over the sequence and print note by note.

A Sequence may also be used to represent how a book is constructed from diverse documents. It is obvious that ordering is important. It may be the case that a specific document is used multiple times within the same book (for example, a specific graphic). Reading the book, you may want to access a specific document by position.

17.4 *Restricted Access Collections*

17.4.1 *Restricted Access Collections Usage Samples*

Deque

A double ended queue may be considered as a sequence with restricted access. It is an ordered collection of elements without a key and no element equality. As there is no element equality, an element value may occur multiple times. There is a first and a last element. You can only add an element as first or last element and only remove the first or the last element from the Deque.

A Deque may be used in the implementation of a pattern matching algorithm where patterns are expressed as regular expressions. Such an algorithm can be described as a non-deterministic finite state machine constructed from the regular expression. The implementation of the regular-pattern matching machine may use a deque to keep track of the states under consideration. Processing a null state requires a stack-like data structure - one of two things to be done is postponed and put at the front of the not being postponed forever list. Processing the other states requires a queue-like data structure, since you do not want to examine a state for the next given character until you are finished with the current character. Combining the two characteristics results in a Deque.

PriorityQueue

A PriorityQueue may be considered as a KeySortedBag with restricted access. It is an ordered collection with zero or more elements. Multiple key values are supported. As no element equality is defined, multiple element values may occur. Access to elements is via key only and sorting is maintained by key. Accessing a PriorityQueue is restricted. You can add an element relative to the ordering relation defined for keys and remove only the first element (e.g., the one with highest priority).

PriorityQueues may be used for implementing a printer queue. A print job's priority may depend on the number of pages, time of queuing, and other characteristics. This priority is the key of the print job. When a user adds a print job it is added relative to its priority. The printer daemon always removes the job with the highest priority from the queue.

PriorityQueues also may be used as special queues in workflow management to prioritize work items.

Queue

A queue may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (enqueue) an element as last element and only remove (dequeue) the first element from the Queue. That is, a queue exposes FIFO behavior.

You would use a queue in tree traversal to implement a breadth first search algorithm.

Queues may be used for the implementation of all kinds of buffered communication where it is important that the receiving side handles messages in the same order as they were sent. Queues may be used in workflow management environments where queues collect messages waiting for processing.

Stack

A Stack may be considered as a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. There is a first and a last element. You can only add (push) an element as last element (at the top) and only remove (pop) the last element from the Stack (from the top). That is, a Stack exposes LIFO behavior. The classical application for a stack is the simulation of a calculator with Reverse Polish Notation. The calculator engine may get an arithmetic expression. Parsing the expression operands are pushed on to the stack. When an operator is encountered, the appropriate number of operands is popped off the stack, the operation performed, and the result pushed on the stack.

A Stack also may be used in the implementation of a window manager to maintain the order in which the windows are superimposed.

17.5 The CosCollection Module

17.5.1 Interface Hierarchies

Collection Interface Hierarchies

The collection interfaces of the Collection Services are organized in two separate hierarchies, as shown in Figure 17-1 on page 17-17 and Figure 17-2 on page 17-17. The inner nodes of the hierarchy may be thought of as abstract views. They represent the basic properties and their combinations. Leaf nodes may be thought of as concrete interfaces for which implementations are provided and from which instances can be created via a collection factory. The organization of the interfaces as a hierarchy enables reuse and the polymorphic usage of the collections from typed languages such as C++.

Each abstract view is defined in terms of operations and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. This view allows adding elements to and iterating over the collection.

In addition to the common collection operations, collections whose elements define equality or key equality provide operations for locating and retrieving elements by a given element or key value.

Ordered collections provide the notion of well-defined explicit positioning of elements, either by element key ordering relation or by positional element access.

Sorted collections provide no further operations, but introduce a new semantics; namely, that their elements are sorted by element or key value. These properties are combined through multiple inheritance.

The fourth property, uniqueness/multiplicity of elements and keys, is not represented by a separate abstract view for combination with other properties. This was done to reduce the complexity of the hierarchy. Instead, operations related to multiplicity are provided in the base interface from which the interface specializations with multiplicity are derived.

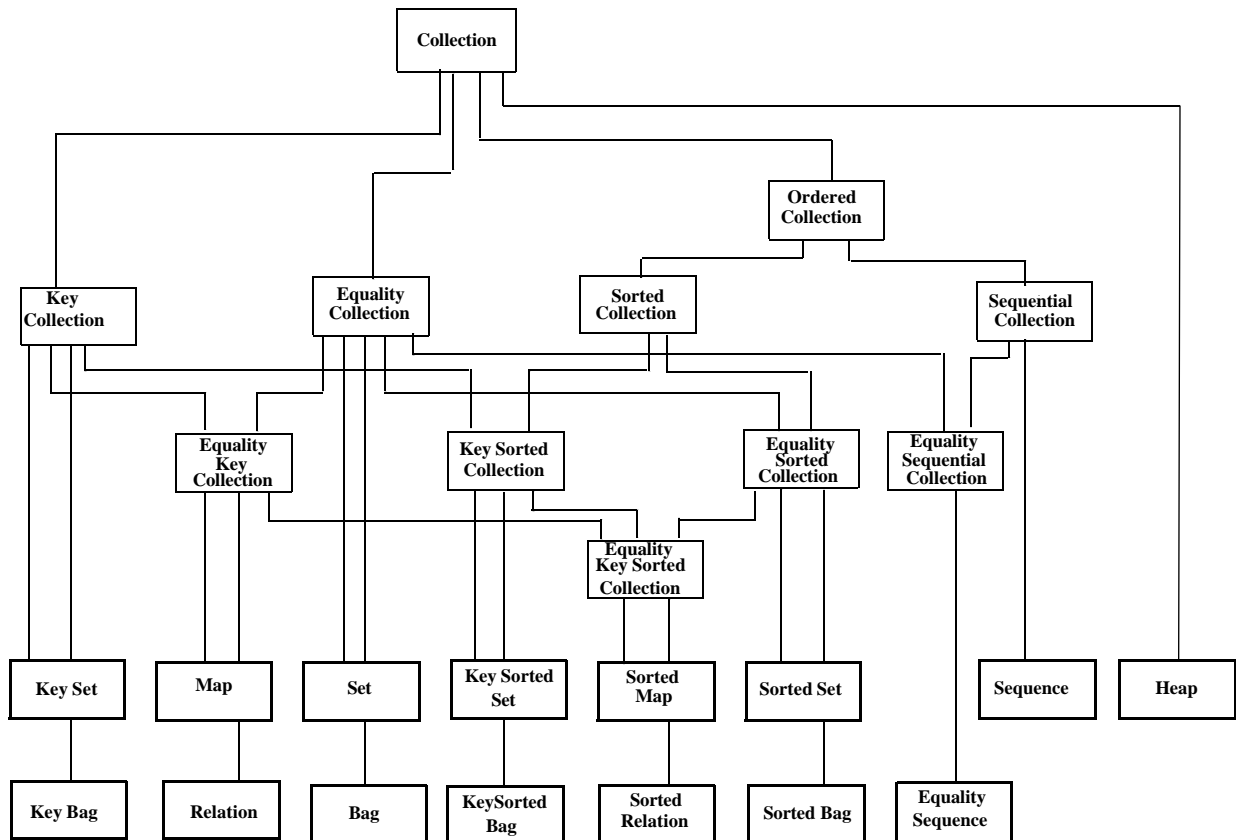


Figure 17-1 Collections Interfaces Hierarchy

The restricted access collections form their own hierarchy as shown in Figure 17-2 on page 17-17. This abstract view defines the operations that all restricted access collections have in common.

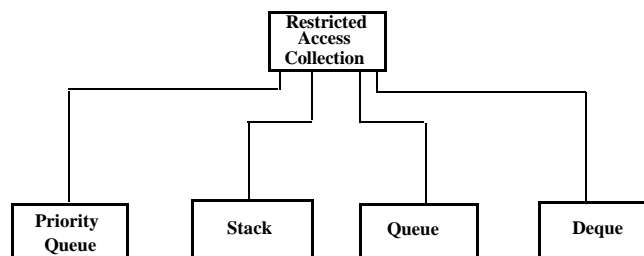


Figure 17-2 Restricted Access Collections Interface Hierarchy

Iterator Hierarchy

The iterator interface hierarchy parallels the Collection interface hierarchy shown in Figure 17-3 on page 17-18. The defined interfaces support the fine-grain processing of very large collections via an iterator only and support a generic programming model similar to what was introduced with ANSI STL to the C++ world. Concepts like constness of iterators, reverse iterators, bulk and combined operations are offered to strengthen the support for the generic programming model.

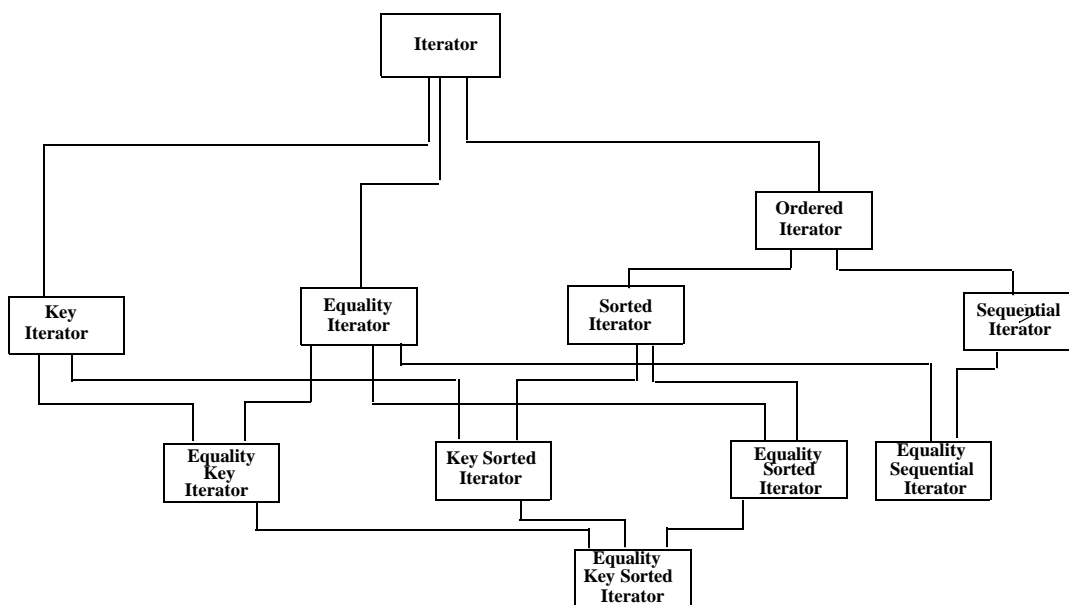


Figure 17-3 Iterator Interface Hierarchy

The top level **Iterator** interface represents a generic iterator that can be used for iteration over and manipulation of all collections independent of their type. The top level iterator allows you to add, retrieve, replace, and remove elements. There are operations to clone, assign, and test iterators for equality. There are tests on the iterator state and you can check whether an iterator is *const*, created for a given collection, or created for the same collection as another iterator.

The **OrderedIterator** interface adds those operations which are useful on collections with an explicit notion of ordering (all those collections inheriting from the **OrderedCollection** interface). An ordered iterator can be moved forward and backward, set to a position, and its position can be computed. Only ordered iterators can be used with “reverse” semantics. The **SequentialIterator** is used with sequentially ordered collections where it is possible to add elements at a user-defined position so that the iterator offers the capability to add elements relative to its position.

The `KeyIterator` and `EqualityIterator` interface add operations for positioning an iterator by key or element value. The sorted versions of these interfaces add respective backward movements and the capability to define lower and upper bounds in sorted collections.

An iterator is always created for a collection using the collection as iterator factory. Each iterator type is supported by each collection type. The Iterators and the Collections that are supported by all interfaces derived from those collections are listed in Table 17-2 on page 17-19.

Table 17-2 Iterators and Collections

	Supported by all interfaces derived from:
Iterator	Collection
<code>OrderedIterator</code>	<code>OrderedCollection</code>
<code>SequentialIterator</code>	<code>SequentialCollection</code>
<code>EqualitySequentialIterator</code>	<code>EqualitySequentialCollection</code>
<code>KeyIterator</code>	<code>KeyCollection</code>
<code>EqualityIterator</code>	<code>EqualityCollection</code>
<code>EqualityKeyIterator</code>	<code>EqualityKeyCollection</code>
<code>SortedIterator</code>	<code>SortedCollection</code>
<code>KeySortedIterator</code>	<code>KeySortedCollection</code>
<code>EqualitySortedIterator</code>	<code>EqualitySortedCollection</code>
<code>EqualityKeySortedIterator</code>	<code>EqualityKeySortedCollection</code>

17.5.2 Exceptions and Type Definitions

The following exceptions are used by the subsequently defined interfaces.

```
module CosCollection {
  // Type definitions
  typedef sequence<any> AnySequence;
  typedef string Istring;
  struct NVPair {Istring name; any value;};
  typedef sequence<NVPair> ParameterList;

  // Exceptions
  exception EmptyCollection{};
```

```
exception PositionInvalid{};
enum IteratorInvalidReason {is_invalid, is_not_for_collection,
is_const};
exception IteratorInvalid {IteratorInvalidReason why;};
exception IteratorInBetween{};
enum ElementInvalidReason {element_type_invalid,
positioning_property_invalid, element_exists};
exception ElementInvalid {ElementInvalidReason why;};
exception KeyInvalid {};
exception ParameterInvalid {unsigned long which; Istring why;};
```

AnySequence

A type definition for a sequence of values of type **any** used in bulk operations.

Istring

A type definition used as place holder for a future IDL internationalized string data type.

ParameterList

A sequence of name-value pairs of type **NVPair** and used as a generic parameter list in a generic collection creation operation.

EmptyCollection

Raised when an operation to remove an element is invoked on an empty collection.

PositionInvalid

Raised when an operation on an ordered collection passes a position out of the allowed range, that is less than 1 or greater than the number of elements in the collections.

IteratorInvalid

Raised when an operation uses an iterator pointing to nothing, that is, using an *invalid* iterator (**in_valid**) or when an operation uses an iterator which was not created for the collection (**is_not_for_collection**) or if one tries to modify a collection via an iterator that is created with **const** designation (**is_const**).

IteratorInBetween

Raised when an operation uses an iterator in a way that does not allow the state *in-between* such as all “..._at” operations.

ElementInvalid

Raised when one of the operations passes an element that is for one of several reasons invalid. It is raised

- when the element is not of the expected element type (`element_type_invalid`).
- if one tries to replace an element by another element changing the positioning property (`positioning_property_invalid`).
- when an element is added to a Map and the key already exists (`element_exists`).

KeyInvalid

Raised when one of the operations passes a key that is not of the expected type.

ParameterInvalid

Raised when a parameter passed to the generic collection creation operation of the generic `CollectionFactory` is invalid.

17.5.3 Abstract Collection Interfaces

The Collection Interface

The `Collection` interface represents the most abstract view of a collection. Operations defined in this top level interface can be supported by all collection interfaces in the hierarchy. Each concrete collection interface offers the appropriate operation semantics dependent on the collection properties. It defines operations for:

- adding elements
- removing elements
- replacing elements
- retrieving elements
- inquiring collection information
- creating iterators

```
// Collection
interface Iterator;
interface Command;

interface Collection {

// element type information
readonly attribute CORBA::TypeCode element_type;
```

```
// adding elements
boolean add_element (in any element) raises (ElementInvalid);
boolean add_element_set_iterator (in any element, in Iterator where)
raises (IteratorInvalid, ElementInvalid);
void add_all_from (in Collection collector) raises (ElementInvalid);

// removing elements
void remove_element_at (in Iterator where) raises (IteratorInvalid,
IteratorInBetween);
unsigned long remove_all ();

// replacing elements
void replace_element_at (in Iterator where, in any element)
raises (IteratorInvalid, IteratorInBetween, ElementInvalid);

// retrieving elements
boolean retrieve_element_at (in Iterator where, out any element)
raises (IteratorInvalid, IteratorInBetween);

// iterating over the collection
boolean all_elements_do (in Command what) ;

// inquiring collection information
unsigned long number_of_elements ();
boolean is_empty ();

// destroying collection
void destroy();

// creating iterators
Iterator create_iterator (in boolean read_only);
};
```

Type checking information

readonly attribute CORBA::TypeCode element_type;

Specifies the element type expected in the collection. See also “The Operations Interface” on page 17-118.

Adding elements

boolean add_element (in any element) raises (ElementInvalid);

Description

Adds an element to the collection. The exact semantics of the add operations depends on the properties of the concrete interface derived from the **Collection** that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception **ElementInvalid** is raised.

Return value

Returns **true** if the element is added.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Side effects

All iterators keep their state.

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid);

Description

Adds an element to the collection and sets the iterator to the added element. The exact semantics of the add operations depends on the properties of the concrete interface derived from the **Collection** that the collection is an instance of.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception **ElementInvalid** is raised.

Return value

Returns **true** if the element is added.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

Side effects

All other iterators keep their state.

`void add_all_from (in Collection elements) raises (ElementInvalid);`

Adds all elements of the given collection to this collection. The elements are added in the iteration order of the given collection and consistent with the semantics of the **add** operation. Essentially, this operation is a sequence of **add** operations.

Removing elements

`void remove_element_at (in Iterator where) raises (IteratorInvalid);`

Description

Removes the element pointed to by the given iterator. The given iterator is set to *in-between*.

Exceptions

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception **IteratorInvalid** is raised.

Side effects

Iterators pointing to the removed element go *in-between*. Iterators which do not point to the removed element keep their state.

`unsigned long void remove_all();`

Description

Removes all elements from the collection.

Return value

Returns the number of elements removed.

Side effects

Iterators pointing to removed elements go *in-between*. All other iterators keep their state.

Replacing elements

void `replace_element_at` (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween, ElementInvalid)

Description

Replaces the element pointed to by the iterator by the given element. The given element must have the same positioning property as the replaced element.

- For collections organized according to element properties such as ordering relation, the replace operation must not change this element property.
- For key collections, the new key must be equal to the key replaced.
- For non-key collections with element equality, the new element must be equal to the replaced element as defined by the element equality relation.

Sequential collections have a user-defined positioning property and heaps do not have positioning properties. Element values in sequences and heaps can be replaced freely.

Exceptions

The given element must not change the positioning property; otherwise, the exception `ElementInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Retrieving elements

boolean `retrieve_element_at` (in Iterator where, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves the element pointed to by the given iterator and returns it via the output parameter element.

Return value

Returns `true` if an element is retrieved.

Exceptions

The given iterator must belong to the collection and must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the `any`. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

Iterating over a collection

`boolean all_elements_do (in Command what);`

Description

Calls the “do_on()” operation of the given `Command` for each element of the collection until the “do_on()” operation returns `false`. The elements are visited in iteration order (see “The Command and Comparator Interface” on page 17-122).

- The “do_on()” operation must not remove elements from or add elements to the collection.
- The “do_on()” operation must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return value

Returns `true` if the “do_on()” operation returns `true` for each element it is applied to.

Inquiring collection information

The collection operations do have preconditions which when violated raise exceptions. There are operations for testing those preconditions to enable the user to avoid raising exceptions.

`unsigned long number_of_elements ();`

Return value

Returns the number of elements contained in the collection.

`boolean is_empty ();`

Return value

Returns `true` if the collection is empty.

Destroying a collection

`void destroy();`

Description

Destroys the collection. This includes:

- removing all elements from the collection
- destroying all iterators created for this collection
- destroying the instance of **Operations** passed at creation time to the collection implementation.

Note – Removing elements in case of objects means removing object references, not destroying the collected objects.

Object references to iterators of the collections become invalid.

Creating iterators

Iterator `create_iterator` (in boolean `read_only`);

Creates and returns an iterator instance for this collection. The type of iterator that is created depends on the interface type of this collection. The following table describes the type of iterator that is created for the type of concrete collection.

Table 17-3 Collection interfaces and the iterator interfaces supported

Ordered	Collection Interfaces	Supported Iterator Interface
	Bag	EqualityIterator
yes	SortedBag	EqualitySortedIterator
yes	EqualitySequence	EqualitySequentialIterator
	Heap	Iterator
	KeyBag	KeyIterator
yes	KeySortedBag	KeySortedIterator
	KeySet	KeyIterator
yes	KeySortedSet	KeySortedIterator
	Map	EqualityKeyIterator
yes	SortedMap	EqualityKeySortedIterator
	Relation	EqualityKeyIterator
yes	Sequence	SequentialIterator

Table 17-3 Collection interfaces and the iterator interfaces supported

yes	SortedRelation	EqualityKeySortedIterator
	Set	EqualityIterator
yes	SortedSet	EqualitySortedIterator
yes	Sequence	SequentialIterator

After creation, the iterator is initialized with the state *invalid*, that is, “pointing to nothing.”

If the given parameter `read_only` is `true`, the iterator is created with `const` designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception `IteratorInvalid`).

Note – Collections serve as factories for *their* iterator instances. An iterator is created in the same address space as the collection for which it is created. An iterator instance can only point to elements of the collection for which it was created.

The OrderedCollection Interface

```
interface OrderedIterator;
// OrderedCollection
interface OrderedCollection: Collection {

    // removing elements
    void remove_element_at_position (in unsigned long position) raises
        (PositionInvalid);
    void remove_first_element () raises (EmptyCollection);
    void remove_last_element () raises (EmptyCollection);

    // retrieving elements
    boolean retrieve_element_at_position (in unsigned long position, out
        any element) raises (PositionInvalid);
    boolean retrieve_first_element (out any element) raises
        (EmptyCollection);
    boolean retrieve_last_element (out any element) raises
        (EmptyCollection);

    // creating iterators
    OrderedIterator create_ordered_iterator(in boolean read_only, in
        boolean reverse_iteration);
};
```

Ordered collections expose the ordering of elements in their interfaces. Elements can be accessed at a position and forward and backward movements are possible (i.e., ordered collection can support ordered iterators). Ordering can be implicitly defined via the ordering relationship of the elements or keys (as in sorted collections) or ordering can be user-controlled (as in sequential collections).

In addition to those inherited from the **Collection** Interface, which all ordered collections have in common, the **OrderedCollection** interface provides operations for

- removing elements,
- retrieving elements, and
- creating ordered iterators.

Removing elements

void remove_element_at_position (in unsigned long position) raises (PositionInvalid);

Description

Removes the element from the collection at a given position. The first element of the collection has position 1.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception **PositionInvalid** is raised. A position is valid if it is greater than or equal to 1 and less than or equal to **number_of_elements()**.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

void remove_first_element () raises (EmptyCollection);

Description

Removes the first element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception **EmptyCollection** is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

void remove_last_element () raises (EmptyCollection);

Description

Removes the last element from the collection.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Side effects

All iterators pointing to the removed element go *in-between*. Iterators that do not point to the removed element keep their state.

Retrieving elements

boolean retrieve_element_at_position (in unsigned long position, out any element) raises (PositionInvalid);

Description

Retrieves the element at the given position in the collection and returns it via the output parameter **element**. Position 1 specifies the first element.

Return value

Returns **true** if an element is retrieved.

Exceptions

The value of "position" must be a valid position in the collection; otherwise, the exception `PositionInvalid` is raised.

boolean retrieve_first_element (out any element) raises (EmptyCollection);

Description

Retrieves the first element in the collection and returns it via the output parameter **element**.

Return value

Returns **true** if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

boolean `retrieve_last_element` (out any element) raises (`EmptyCollection`);

Description

Retrieves the last element in the collection and returns it via the output parameter element.

Return value

Returns **true** if an element is retrieved.

Exceptions

The collection must not be empty; otherwise, the exception `EmptyCollection` is raised.

Creating iterators

`OrderedIterator create_ordered_iterator` (in boolean `read_only`, in boolean `reverse_iteration`);

Description

Creates and returns an ordered iterator instance for this collection.

Which type of ordered iterator actually is created depends on the interface type of this collection. Table 17-1 on page 17-4 describes which type of ordered iterator is created for which type of concrete ordered collection.

After creation, the iterator is initialized with the state invalid, that is, “pointing to nothing.”

Exceptions

If the given parameter `read_only` is **true**, the iterator is created with `const` designation (i.e., a trial to modify the collection content via this iterator is rejected and raises the exception `IteratorInvalid`).

Side effects

If the given parameter `reverse_iteration` is **true**, the iterator is created with reverse iteration semantics. Only ordered iterators can be created with reverse semantics.

The SequentialCollection Interface

```
interface Comparator;
interface SequentialCollection: OrderedCollection {
    // adding elements
    void add_element_as_first (in any element) raises (ElementInvalid);
```

```
void add_element_as_first_set_iterator (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_last (in any element) raises (ElementInvalid);
void add_element_as_last_set_iterator (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);

void add_element_as_next (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);
void add_element_as_previous (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);
void add_element_at_position (in unsigned long position, in any
element) raises (PositionInvalid, ElementInvalid);
void add_element_at_position_set_iterator (in unsigned long
position, in any element, in Iterator where) raises
(PositionInvalid, ElementInvalid, IteratorInvalid);

// replacing elements
void replace_element_at_position (in unsigned long position, in any
element) raises (PositionInvalid, ElementInvalid);
void replace_first_element (in any element) raises (ElementInvalid,
EmptyCollection);
void replace_last_element (in any element) raises (ElementInvalid,
EmptyCollection);

// reordering elements
void sort (in Comparator comparison);
void reverse();
};
```

Sequential collections expose user-controlled sequential ordering. Determine where elements are added by comparing to sorted collections where the “where an element is added” is determined implicitly by the defined element or key comparison.

The `SequentialCollection` interface adds all those operations to the `OrderedCollection` interface. “The `SequentialCollection` Interface” on page 17-31 describes operators that are unique for positional element access for

- adding elements,
- replacing elements, and
- re-ordering elements.

Adding elements

```
void add_element_as_first (in any element) raises (ElementInvalid);
```

Description

Adds the element to the collection as the first element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_first_set_iterator` (in any element, in Iterator where)
raises (`ElementInvalid`,`IteratorInvalid`);

Description

Adds the element to the collection as the first element in sequential order and sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_last` (in any element) raises (`ElementInvalid`);

Description

Adds the element to the collection as the last element in sequential order.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_last_set_iterator` (in any element, in Iterator where)
raises (`ElementInvalid`,`IteratorInvalid`);

Description

Adds the element to the collection as the last element in sequential order. Sets the iterator to the added element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All other iterators keep their state.

`void add_element_as_next(in any element, in Iterator where)` raises (`ElementInvalid`, `IteratorInvalid`);

Description

Adds the element to the collection after the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between*, the element is added before the iterator's "potential next" element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and be valid; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_as_previous (in any element, in Iterator where)` raises (`IteratorInvalid`, `ElementInvalid`);

Description

Adds the element to the collection as the element previous to the element pointed to by the given iterator. Sets the iterator to the added element. If the iterator is in the state *in-between*, the element is added after the iterator's "potential previous" element.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_at_position` (in unsigned long position, in any element)
raises(`PositionInvalid`, `ElementInvalid`);

Description

Adds the element at the given position to the collection. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to `number_of_elements() + 1`); otherwise, the exception `PositionInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`void add_element_at_position_set_iterator` (in unsigned long position, in any element, in `Iterator` where) raises (`PositionInvalid`, `ElementInvalid` `IteratorInvalid`);

Description

Adds the element at the given position to the collection and sets the iterator to the added element. If an element exists at the given position, the new element is added as the element preceding the existing element.

Exceptions

The position must be valid (i.e., greater than or equal to 1 and less than or equal to `number_of_elements() + 1`); otherwise, the exception `PositionInvalid` is raised.

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

Replacing elements

void replace_element_at_position (in unsigned long position, in any element) raises (PositionInvalid, ElementInvalid);

Description

Replaces the element at a given position with the given element. The given position must be valid (i.e., greater than or equal to 1 and less than or equal to number_of_elements()).

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

void replace_first_element (in any element) raises (ElementInvalid, EmptyCollection);

Description

Replaces the first element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

void replace_last_element (in any element) raises (ElementInvalid, EmptyCollection);

Description

Replaces the last element with the given element.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The collection must not be empty; otherwise, the exception EmptyCollection is raised.

Re-ordering elements

void sort (in Comparator comparison);

Description

Sorts the collection so that the elements occur in ascending order. The relation of two elements is defined by the “compare” method, which a user provides when implementing an interface derived from Comparator. See “The Command and Comparator Interface” on page 17-122.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

void reverse ();

Description

Orders elements in reverse order.

Side effects

All iterators in the state *in-between* go *invalid*.

All other iterators keep their state.

The SortedCollection Interface

```
interface SortedCollection: OrderedCollection{}
```

Sorted collections currently do not provide further operations but define a more specific behavior; namely, that the elements or their keys are sorted with respect to a user-defined element or key compare. See “The OrderedCollection Interface” on page 17-28.

The EqualityCollection Interface

```
interface EqualityCollection: Collection {
```

```
    // testing element containment
```

```
    boolean contains_element (in any element) raises(ElementInvalid);
```

```
    boolean contains_all_from (in Collection collector)
    raises(ElementInvalid);
```

```
    // adding elements
```

```
boolean locate_or_add_element (in any element) raises
(ElementInvalid);

boolean locate_or_add_element_set_iterator (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);

// locating elements

boolean locate_element (in any element, in Iterator where) raises (
ElementInvalid, IteratorInvalid);

boolean locate_next_element (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);

boolean locate_next_different_element (in Iterator where) raises
(IteratorInvalid, IteratorInBetween);

// removing elements

boolean remove_element (in any element) raises (ElementInvalid);

unsigned long remove_all_occurrences (in any element) raises
(ElementInvalid);

// inquiring collection information

unsigned long number_of_different_elements ();

unsigned long number_of_occurrences (in any element)
raises(ElementInvalid);

};
```

Collections whose elements define equality introduce operations which exploit the defined element equality. These operations are for finding elements by element value (and adding if not found), for testing containment of a given element, and inquiring the collection about how many elements of a given value were collected.

Testing element containment

```
boolean contains_element (in any element) raises (ElementInvalid);
```

Return value

Returns **true** if the collection contains an element equal to the given element.

Exceptions

The given elements must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

```
boolean contains_all_from (in Collection collector) raises (ElementInvalid);
```

Return value

Returns **true** if all the elements of the given collection are contained in the collection. The definition of containment is given in “contains_element.”

Exceptions

The elements in the given collection must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Adding elements

`boolean locate_or_add_element` (in any element) raises (`ElementInvalid`);

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in `add`.

Return value

Returns **true** if the element was found.

Returns **false** if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

`boolean locate_or_add_element_set_iterator` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates an element in the collection that is equal to the given element. If no such element is found, the element is added as described in `add`. The iterator is set to the found or added element.

Return value

Returns **true** if the element was found.

Returns **false** if the element had to be added.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All other iterators keep their state.

Locating elements

boolean `locate_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns `true` if an element is found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

boolean `locate_next_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the located element. The iterator is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited. If the iterator is in the state *in-between*, locating is started at the iterator's "potential next" element.

Return value

Returns **true** if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The iterator must belong to the collection and must be valid; otherwise, the exception **IteratorInvalid** is raised.

boolean locate_next_different_element (in **Iterator** where) raises (**IteratorInvalid**, **IteratorInBetween**);

Description

Locates the next element in iteration order that is different from the element pointed to by the given iterator. If no more elements are left to be visited, the given iterator will no longer be valid.

Return value

Returns **true** if the next different element was found.

Exception

The iterator must belong to the collection and point to an element of the collection; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

Removing elements

boolean remove_element (in any element) raises (**ElementInvalid**);

Description

Removes an element in the collection that is equal to the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of the given element will be removed.

Return value

Returns **true** if an element was removed.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Side effects

If an element was removed, all iterators pointing to this element go *in-between*.

All other iterators keep their state.

unsigned long remove_all_occurrences (in any element) raises (ElementInvalid);

Description

Removes all elements from the collection that are equal to the given element and returns the number of elements removed.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

Side effects

All iterators pointing to elements removed go *in-between*.

All iterators keep their state.

Inquiring collection information

unsigned long number_of_different_elements ();

Return value

Returns the number of different elements in the collection.

unsigned long number_of_occurrences (in any element) raises (ElementInvalid);

Return value

Returns the number of occurrences of the given element in the collection.

Exceptions

The given element must be of the expected type; otherwise, the exception ElementInvalid is raised.

The KeyCollection Interface

```
interface KeyCollection: Collection {  
    // Key type information  
    readonly attribute CORBA::TypeCode key_type;  
  
    // testing containment
```

```

boolean contains_element_with_key (in any key) raises(KeyInvalid);
boolean contains_all_keys_from (in KeyCollection collector)
raises(KeyInvalid);

// adding elements
boolean locate_or_add_element_with_key (in any element)
raises(ElementInvalid);
boolean locate_or_add_element_with_key_set_iterator (in any
element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);

// adding or replacing elements
boolean add_or_replace_element_with_key (in any element)
raises(ElementInvalid);
boolean add_or_replace_element_with_key_set_iterator (in any
element, in Iterator where) raises (ElementInvalid,
IteratorInvalid);

// removing elements
boolean remove_element_with_key(in any key) raises(KeyInvalid);
unsigned long remove_all_elements_with_key (in any key)
raises(KeyInvalid);

// replacing elements
boolean replace_element_with_key (in any element)
raises(ElementInvalid);
boolean replace_element_with_key_set_iterator (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);

// retrieving elements
boolean retrieve_element_with_key (in any key, out any element)
raises (KeyInvalid);

// computing the keys
void key (in any element, out any key) raises (ElementInvalid);
void keys (in AnySequence elements, out AnySequence keys) raises
(ElementInvalid);

// locating elements
boolean locate_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
boolean locate_next_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
boolean locate_next_element_with_different_key (in Iterator where)
raises (IteratorInBetween, IteratorInvalid);

```

```
// inquiring collection information
unsigned long number_of_different_keys ();
unsigned long number_of_elements_with_key (in any key)
raises(KeyInvalid);
};
```

A **KeyCollection** is a collection which offers associative access to its elements via a key. All elements of such a collection are keyed elements (i.e., they do have a key which is computed from the element value). How to compute the key from an element value is user-defined. A user specializes the **Operations** interface and implements the operation **key()** as desired (see “The Operations Interface” on page 17-118). This information is passed to the collection at creation time.

Type checking information

readonly attribute CORBA::TypeCode key_type;

Specifies the key type expected in the collection. See also “The Operations Interface” on page 17-118.

Testing containment

boolean contains_element_with_key (in any key) raises (KeyInvalid);

Return value

Returns **true** if the collection contains an element with the same key as the given key.

Exceptions

The given key has to be of the expected type; otherwise, the exception **KeyInvalid** is raised.

boolean contains_all_keys_from (in KeyCollection collector) raises(KeyInvalid);

Return value

Returns **true** if all of the keys of the given collection are contained in the collection.

Exceptions

The keys of the given collection have to be of the expected type of this collection; otherwise, the exception **KeyInvalid** is raised.

Adding elements

boolean `locate_or_add_element_with_key` (in any element)
 raises(`ElementInvalid`);

Description

Locates an element with the same key as the key in the given element. If no such element exists the element is added; otherwise, the collection remains unchanged.

Return value

Returns `true` if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All iterators keep their state.

boolean `locate_or_add_element_with_key_set_iterator` (in any element, in iterator where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates an element with the same key as the key in the given element and sets the iterator to the located elements (see `locate_element_with_key()`). If no such element exists, the element is added and the iterator is set to the element added.

Return value

Returns `true` if the element is located.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Side effects

All iterators keep their state.

boolean `add_or_replace_element_with_key` (in any element) raises (`ElementInvalid`);

Description

If the collection contains an element with the key equal to the key in the given element, the element is replaced with the given element; otherwise, the given element is added to the collection.

Return value

Returns **true** if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Side effects

All iterators keep their state.

boolean **add_or_replace_element_with_key_set_iterator** (in any element, in iterator where) raises (**ElementInvalid**, **IteratorInvalid**);

Description

If the collection contains an element with the key equal to the key in the given element, the iterator is set to that element and the element is replaced with the given element; otherwise, the given element is added to the collection, and the iterator set to the added element.

Return value

Returns **true** if the element was added.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

Side effects

All iterators keep their state.

Removing elements

boolean **remove_element_with_key** (in any key) raises (**KeyInvalid**);

Description

Removes an element from the collection with the same key as the given key. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be removed.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

Side effects

If an element was removed, all iterators pointing to the element go *in-between*.

All other iterators keep their state.

unsigned long `remove_all_elements_with_key` (in any key) raises(`KeyInvalid`);

Description

Removes all elements from the collection with the same key as the given key.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

Side effects

Iterators pointing to elements removed go *in-between*.

All other iterators keep their state.

Replacing elements

boolean `replace_element_with_key` (in any element) raises (`ElementInvalid`);

Description

Replaces an element with the same key as the given element by the given element. If no such element exists, the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns `true` if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

boolean `replace_element_with_key_set_iterator` (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Replaces an element with the same key as the given element by the given element, and sets the iterator to this element. If no such element exists, the iterator is invalidated and the collection remains unchanged. In collections with non-unique elements, an arbitrary occurrence of such an element will be replaced.

Return value

Returns `true` if an element was replaced.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

Computing keys

void `key` (in any element, out any key) raises(`ElementInvalid`);

Description

Computes the key of the given element and returns it via the output parameter `key`.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

void `keys` (in Any Sequence elements, out Any Sequence keys)
raises(`ElementInvalid`);

Description

Computes the keys of the given elements and returns them via the output parameter `keys`.

Exceptions

The given elements must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

An implementation may rely on the key operation of a user supplied interface derived from **Operations**. An instance of this interface is passed to a collection at creation time and can be used in the collection implementation.

Locating elements

boolean `locate_element_with_key` (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to point to the element in the collection, or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

boolean `locate_next_element_with_key` (in any key, in Iterator where) raises (KeyInvalid, IteratorInvalid);

Description

Locates the next element in iteration order with the key equal to the given key, starting at the element next to the one pointed to by the given iterator. Sets the iterator to point to the element in the collection. The given iterator is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited. If the iterator is in the *in-between* state, locating starts at the iterator's "potential next" element.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must belong to the collection and must be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean locate_next_element_with_different_key (in Iterator where)
raises(IteratorInvalid, IteratorInBetween)`

Description

Locates the next element in the collection in iteration order with a key different from the key of the element pointed to by the given iterator. If no such element exists, the given iterator is no longer valid.

Return value

Returns `true` if an element was found.

Exceptions

The given iterator must belong to the collection and must point to an element; otherwise, the exception `IteratorInvalid` respectively `IteratorInBetween` is raised.

Inquiring collection information

`unsigned long number_of_different_keys ();`

Return value

Returns the number of different keys in the collection.

`unsigned long number_of_elements_with_key (in any key) raises(KeyInvalid);`

Return value

Returns the number elements with key specified.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The EqualityKeyCollection Interface

`interface EqualityKeyCollection : EqualityCollection, KeyCollection{;`

Description

This interface combines the interfaces representing the properties “key access” and “element equality.” See “The EqualityCollection Interface” on page 17-37 and “The KeyCollection Interface” on page 17-42.

The KeySortedCollection Interface

```
interface KeySortedCollection : KeyCollection, SortedCollection {
    // locating elements
    boolean locate_first_element_with_key (in any key, in Iterator
    where) raises (KeyInvalid, IteratorInvalid);
    boolean locate_last_element_with_key (in any key, in Iterator where)
    raises (KeyInvalid, IteratorInvalid);
    boolean locate_previous_element_with_key (in any key, in Iterator
    where) raises (KeyInvalid, IteratorInvalid);
    boolean locate_previous_element_with_different_key (in Iterator
    where) raises (IteratorInBetween, IteratorInvalid);
};
```

This interface combines the interfaces representing the properties “key access” and “ordering.” See “The KeyCollection Interface” on page 17-42 and “The SortedCollection Interface” on page 17-37.

Locating elements

```
boolean locate_first_element_with_key (in any key, in Iterator where)
raises (KeyInvalid, IteratorInvalid);
```

Description

Locates the first element in iteration order in the collection with the same key as the given key. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

```
boolean locate_last_element_with_key (in any key, in Iterator where) raises
(KeyInvalid, IteratorInvalid);
```

Description

Locates the last element in iteration order in the collection with the same key as the given key. Sets the given iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

boolean `locate_previous_element_with_key` (in any key, in Iterator where) raises (**KeyInvalid**, **IteratorInvalid**);

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, locating begins at the iterator's "potential previous" element.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

The given iterator must belong to the collection and be valid; otherwise, the exception **IteratorInvalid** is raised.

boolean `locate_previous_element_with_different_key`(in Iterator where) raises (**IteratorInBetween**, **IteratorInvalid**);

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The given key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

The given iterator must point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

The EqualitySortedCollection Interface

This interface combines the interfaces representing the properties “element equality” and “ordering.” See “The EqualityCollection Interface” on page 17-37 and “The SortedCollection Interface” on page 17-37. It adds those methods which exploit the combination of both properties.

```
interface EqualitySortedCollection : EqualityCollection,
SortedCollection {
  // locating elements
  boolean locate_first_element (in any element, in Iterator where)
  raises (ElementInvalid, IteratorInvalid);
  boolean locate_last_element (in any element, in Iterator where)
  raises (ElementInvalid, IteratorInvalid);
  boolean locate_previous_element (in any element, in Iterator where)
  raises
  raises (ElementInvalid, IteratorInvalid);
  boolean locate_previous_different_element (in Iterator where) raises
  (IteratorInvalid);
};
```

Locating elements

```
boolean locate_first_element (in any element, in Iterator where) raises
(ElementInvalid, IteratorInvalid);
```

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean `locate_last_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean `locate_previous_element` (in any element, in `Iterator` where) raises (`ElementInvalid`, `IteratorInvalid`);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns `true` if an element was found.

Exceptions

The given element must be of the expected type otherwise the exception `ElementInvalid` is raised.

The given iterator must belong to the collection; otherwise, the exception `IteratorInvalid` is raised.

boolean locate_previous_different_element (in Iterator where) raises (IteratorInBetween, IteratorInvalid);

Description

Locates the previous element in iteration order with a value different from the element pointed to, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The given iterator must point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

The EqualityKeySortedCollection Interface

```
interface EqualityKeySortedCollection: EqualityCollection, KeyCollection,
SortedCollection {};
```

This interface combines the interface representing the properties “element equality,” “key access,” and “ordering.”

The EqualitySequentialCollection Interface

This interface combines the interface representing the properties “element equality” and “(sequential) ordering” and offers additional operations which exploit this combination.

```
interface EqualitySequentialCollection: EqualityCollection,
SequentialCollection
{
    // locating elements
    boolean locate_first_element_with_value (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);
    boolean locate_last_element_with_value (in any element, in Iterator
where) raises (ElementInvalid, IteratorInvalid);
    boolean locate_previous_element_with_value (in any element, in
Iterator where) raises (ElementInvalid, IteratorInvalid);
};
```

Locating elements

boolean `locate_first_element_with_value` (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);

Description

Locates the first element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The given iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

boolean `locate_last_element_with_value` (in any element, in Iterator where)
raises (ElementInvalid, IteratorInvalid);

Description

Locates the last element in iteration order in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The iterator must belong to the collection; otherwise, the exception **IteratorInvalid** is raised.

boolean `locate_previous_element_with_value` (in any element, in Iterator where) raises (ElementInvalid, IteratorInvalid);

Description

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, locating begins at the iterators “potential previous” element.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

The iterator must belong to the collection and be valid; otherwise, the exception **IteratorInvalid** is raised.

17.5.4 Concrete Collections Interfaces

The previously listed “abstract views” on collections combine the properties “key access,” “element equality,” and “ordering relationship” on elements. The subsequent interfaces add “uniqueness” support for “multiples.” To reduce the complexity of the hierarchy, this fourth property is not represented by a separate interface.

The KeySet Interface

```
interface KeySet: KeyCollection {};
```

The **KeySet** offers an interface representing the property “key access” with the semantics of “unique keys required.” See “The KeyCollection Interface” on page 17-42.

The KeyBag Interface

```
interface KeyBag: KeyCollection {};
```

The **KeyBag** offers the interface representing the property “key access” with multiple keys allowed. See “The KeyCollection Interface” on page 17-42.

The Map Interface

```
interface Map : EqualityKeyCollection {
  // set theoretical operations
  void difference_with (in Map collector) raises (ElementInvalid);
  void add_difference (in Map collector1, in Map collector2) raises
    (ElementInvalid);
```

```
void intersection_with (in Map collector) raises (ElementInvalid);
void add_intersection (in Map collector1, in Map collector2) raises
(ElementInvalid);
void union_with (in Map collector) raises (ElementInvalid);
void add_union (in Map collector1, in Map collector2) raises
(ElementInvalid);

// testing equality
boolean equal (in Map collector) raises (ElementInvalid);
boolean not_equal (in Map collector) raises (ElementInvalid);
};
```

The Map offers the interface representing the combination of the properties “element equality testable” and “key access” and supports the semantics “unique keys required” (which implies unique elements). See “The EqualityKeyCollection Interface” on page 17-50.

With element equality defined, a test on equality for collections of the same type is possible as well as a meaningful definition of the set theoretical operations.

Set theoretical operations

```
void difference_with (in Map collector) raises (ElementInvalid);
```

Description

Makes this collection the difference between this collection and the given collection. The difference of A and B (A minus B) is the set of elements that are contained in A but not in B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the difference of P and Q contains the element X m-n times if “m > n,” and zero times if “m ≤ n.”

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception **ElementInvalid** is raised.

Side effects

Valid iterators pointing to removed elements go *in-between*. All other iterators keep their state.

```
void add_difference (in Map collector1, in Map collector2) raises
(ElementInvalid);
```

Description

Creates the difference between the two given collections and adds the difference to this collection.

Exceptions

Elements of the given collections must be of the expected type in this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding the difference takes place one by one so the semantics for `add` applies here for raised exceptions and iterator state.

`void intersection_with (in Map collector) raises (ElementInvalid);`

Description

Makes this collection the intersection of this collection and the given collection. The intersection of A and B is the set of elements that is contained in both A and B.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the intersection of P and Q contains the element X “MIN(m,n)” times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Valid iterators of this collection pointing to removed elements go *in-between*.

All other iterators keep their state.

`void add_intersection (in Map collector1, in Map collector2) raises (ElementInvalid);`

Description

Creates the intersection of the two given collections and adds the intersection to this collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

Side effects

Adding the intersection takes place one by one so the semantics for **add** apply here for raised exceptions and iterator state.

void union_with (in Map collector) raises (ElementInvalid);

Description

Makes this collection the union of this collection and the given collection. The union of A and B are the elements that are members of A or B or both.

The same operation is defined for other collections, too. The following rule applies for collections with multiple elements: If collection P contains the element X m times and collection Q contains the element X n times, the union of P and Q contains the element X m+n times.

Exceptions

Elements of the given collection must have the expected type of this collection; otherwise, the exception **ElementInvalid** is raised.

Side effects

Adding takes place one by one so the semantics for **add** applies here for raised exceptions and iterator state.

void add_union (in Map collector1, in Map collector2) raises (ElementInvalid);

Description

Creates the union of the two given collections and adds the union to the collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception **ElementInvalid** is raised.

Side effects

Adding the intersection takes place one by one; therefore, the semantics for **add** applies here for validity of iterators and raised exceptions.

Testing equality

boolean equal (in Map collector) raises (ElementInvalid);

Return value

Returns **true** if the given collection is equal to the collection.

This operation is defined for other collections, too. Two collections are equal if the number of elements in each collection is the same and if the following conditions (depending on the collection properties) are fulfilled.

- **Collections with unique elements:** If the collections have unique elements, any element that occurs in one collection must occur in the other collections, too.
- **Collections with non-unique elements:** If an element has *n* occurrences in one collection, it must have exactly *n* occurrences in the other collection.
- **Sequential collections:** They are sequential collections if they are lexicographically equal based on element equality defined for the elements of the sequential collection.

Exceptions

Elements of the given collections must have the expected type of this collection; otherwise, the exception `ElementInvalid` is raised.

`boolean not_equal` (in `Map` collector) raises (`ElementInvalid`);

Return value

Returns `true` if the given collection is not equal to this collection.

The Relation Interface

```
interface Relation : EqualityKeyCollection {
  // equal, not_equal, and the set-theoretical operations as defined
  for Map
};
```

The `Relation` interface offers the interface representing the combination of the properties “element equality testable” and “key access” and supports the semantics “multiple elements allowed.” See “The `EqualityKeyCollection` Interface” on page 17-50. For a definition of the set-theoretical operation see “The `Map` Interface” on page 17-57.

The Set Interface

```
interface Set : EqualityCollection {
  // equal, not_equal, and the set theoretical operations as defined
  for Map
};
```

The `Set` offers the interface representing the property “element equality testable” with the semantics of “unique elements required.” See “The `EqualityCollection` Interface” on page 17-37.

The Bag Interface

```
interface Bag : EqualityCollection {  
    // equal, not_equal, and the set theoretical operations as defined  
    // for Map  
};
```

The **Bag** offers the interface representing the property “element equality testable” with the semantics of “multiples allowed.” See “The EqualityCollection Interface” on page 17-37.

The KeySortedSet Interface

```
interface KeySortedSet : KeySortedCollection {  
    long compare (in KeySortedSet collector, in Comparator comparison);  
};
```

The **KeySortedSet** offers the sorted variant of **KeySet**. See “The KeySortedCollection Interface” on page 17-51.

The sorted variant of **KeySet** introduces a new operation **compare** which can be supported only when there is “ordering.” This operation takes an instance of a user-defined **Comparator** as given parameter. See “The Command and Comparator Interface” on page 17-122.

The **Comparator** defines the comparison to be used for the elements in the context of this **compare** operation. Comparison on two **KeySortedSets** then is a lexicographical comparison based on this element comparison.

long compare (in KeySortedSet collector, in Comparator comparison) raises (**ElementInvalid**);

Description

Compares this collection with the given collection. Comparison yields:

- <0 if this collection is less than the given collection,
- 0 if the collection is equal to the given collection, and
- >0 if the collection is greater than the given collection.

Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. If such a pair does not exist, the collection with more elements is the greater one.

The “compare” operation of the user’s comparator (interface derived from **Comparator**) must return a result according to the following rules:

>0	if (element1 > element2)
0	if (element1 = element2)

<0 if (element1 < element2)

Return value

Returns the result of the collection comparison.

The KeySortedBag Interface

```
interface KeySortedBag : KeySortedCollection {
  long compare (in KeySortedBag collector, in Comparator comparison);
};
```

The `KeySortedBag` is the sorted variant of the `KeyBag`. See “The `KeySortedCollection` Interface” on page 17-51. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedMap Interface

```
interface SortedMap : EqualityKeySortedCollection {
  // equal, not_equal, and the set theoretical operations
  long compare (in SortedMap collector, in Comparator comparison);
};
```

The `SortedMap` interface is the sorted variant of a `Map`. See “The `EqualityKeySortedCollection` Interface” on page 17-55. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedRelation Interface

```
interface SortedRelation : EqualityKeySortedCollection {
  // equal, not_equal, and the set theoretical operations
  long compare (in SortedRelation collector, in Comparator
  comparison);
};
```

The `SortedRelation` interface is the sorted variant of a `Relation`. See “The `EqualitySortedCollection` Interface” on page 17-53. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedSet Interface

```
interface SortedSet : EqualitySortedCollection {
  // equal, not_equal, and the set theoretical operations
  long compare (in SortedSet collector, in Comparator comparison);
};
```

The `SortedSet` interface is the sorted variant of a `Set`. The additional operation `compare` is offered. See “The `KeySortedSet` Interface” on page 17-62.

The SortedBag Interface

```
interface SortedBag: EqualitySortedCollection {  
    // equal, not_equal, and the set theoretical operations  
    long compare (in SortedBag collector, in Comparator comparison);  
};
```

The **SortedBag** interface is the sorted variant of a Bag. See “The EqualitySortedCollection Interface” on page 17-53. The additional operation **compare** is offered. See “The KeySortedSet Interface” on page 17-62.

The Sequence Interface

```
interface Sequence : SequentialCollection {  
    // Comparison  
    long compare (in Sequence collector, in Comparator comparison);  
};
```

The **Sequence** supports the interface representing the property “sequential ordering.” This property enables the definition of comparison on two Sequences; therefore, the operation **compare** is offered. See “The SequentialCollection Interface” on page 17-31.

The EqualitySequence Interface

```
interface EqualitySequence : EqualitySequentialCollection {  
    // test on equality  
    boolean equal (in EqualitySequence collector);  
    boolean not_equal (in EqualitySequence collector);  
    // comparison  
    long compare (in EqualitySequence collector, in Comparator  
comparison);  
};
```

The **EqualitySequence** supports the combination of the properties “sequential ordering” and “element equality testable.” See “The EqualitySequentialCollection Interface” on page 17-55. This allows the operations **equal**, **not_equal** and **compare**.

The Heap Interface

```
interface Heap : Collection {};
```

The **Heap** does not support any property at all. It just delivers the basic **Collection** interface. See “The Collection Interface” on page 17-21.

17.5.5 Restricted Access Collection Interfaces

Common data structures, such as a stack, may restrict access to the elements of a collection. The restricted access collections support these data structures. **Stack**, **Queue**, and **Deque** are essentially restricted access Sequences. **PriorityQueue** is essentially a restricted access **KeySortedBag**. For convenience, these interfaces offer the commonly used operation names such as **push**, **pop**, etc. rather than **add_element**, **remove_element_at**. Although the restricted access collections form their own hierarchy, the naming was formed in a way that allows mixing-in with the hierarchy of the combined property collections.

This may be useful to support several views on the same instance of a collection. For example, a “user view” to a job queue with restricted access of a **PriorityQueue** and an “administrator view” to the same print job queue with the full capabilities of a **KeySortedBag**.

17.5.6 Abstract RestrictedAccessCollection Interface

The RestrictedAccessCollection Interface

```
// Restricted Access Collections
interface RestrictedAccessCollection {

    // getting information on collection state
    boolean unfilled ();
    unsigned long size ();

    // removing elements
    void purge ();
};
```

boolean unfilled ();

Return value

Returns **true** if the collection is empty.

unsigned long size ();

Return value

Returns the number of elements in the collection.

void purge ();

Description

Removes all elements from the collection. See “The Collection Interface” on page 17-21.

17.5.7 Concrete Restricted Access Collection Interfaces

The Queue Interface

```
interface Queue : RestrictedAccessCollection {  
  
    // adding elements  
    void enqueue (in any element) raises (ElementInvalid);  
  
    // removing elements  
    void dequeue () raises (EmptyCollection);  
    boolean element_dequeue (out any element) raises (EmptyCollection);  
};
```

A **Queue** may be considered as a restricted access **Sequence**. Elements are added at the end of the queue only and removed from the beginning of the queue. FIFO behavior is delivered.

Adding elements

```
void enqueue (in any element) raises (ElementInvalid);
```

Description

Adds the element as last element to the Queue.

Exceptions

The given element must be the expected type; otherwise, the exception **ElementInvalid** is raised.

Removing elements

```
void dequeue () raises (EmptyCollection);
```

Description

Removes the first element from the queue.

Exceptions

The queue must not be empty; otherwise, the exception **EmptyCollection** is raised.

boolean element_dequeue(out any element) raises (EmptyCollection);

Description

Retrieves the first element in the queue, returns it via the output parameter **element**, and removes it from the queue.

Return value

Returns **true** if an element was retrieved.

Exceptions

The queue must not be empty; otherwise, the exception **EmptyCollection** is raised.

The Dequeue Interface

```
interface Deque : RestrictedAccessCollection {

    // adding elements
    void enqueue_as_first (in any element) raises (ElementInvalid);
    void enqueue_as_last (in any element) raises (ElementInvalid);

    // removing elements
    void dequeue_first () raises (EmptyCollection);
    boolean element_dequeue_first (out any element) raises
        (EmptyCollection);
    void dequeue_last () raises (EmptyCollection);
    boolean element_dequeue_last (out any element) raises
        (EmptyCollection);
};
```

The **Deque** may be considered as a restricted access **Sequence**. Adding and removing elements is only allowed at both ends of the double-ended queue. The semantics of the **Deque** operation is comparable to the operations described for the **Queue** interface. See “The Queue Interface” on page 17-66.

The Stack Interface

```
interface Stack: RestrictedAccessCollection {

    // adding elements
    void push (in any element) raises (ElementInvalid);

    // removing and retrieving elements
    void pop () raises (EmptyCollection);
    boolean element_pop (out any element) raises (EmptyCollection);
```

```
boolean top (out any element) raises (EmptyCollection);  
};
```

The **Stack** may be considered as a restricted access **Sequence**. Adding and removing elements is only allowed at the end of the queue. LIFO behavior is delivered.

Adding elements

```
void push (in any element) raises (ElementInvalid);
```

Description

Adds the element to the stack as the last element.

Exceptions

The given element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

Removing elements

```
void pop () raises (EmptyCollection);
```

Description

Removes the last element from the stack.

Exceptions

The stack must not be empty; otherwise, the exception **EmptyCollection** is raised.

```
boolean element_pop (out any element) raises (EmptyCollection);
```

Description

Retrieves the last element from the stack and returns it via the output parameter **element** and removes it from the stack.

Return value

Returns **true** if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception **EmptyCollection** is raised.

Retrieving elements

```
boolean top (out any element) raises (EmptyCollection);
```

Description

Retrieves the last element from the stack and returns it via the output parameter `element`.

Return value

Returns `true` if an element is retrieved.

Exceptions

The stack must not be empty; otherwise, the exception `EmptyCollection` is raised.

The PriorityQueue Interface

```
interface PriorityQueue: RestrictedAccessCollection {
  // adding elements
  void enqueue (in any element) raises (ElementInvalid);

  // removing elements
  void dequeue () raises (EmptyCollection);
  boolean element_dequeue (out any element) raises (EmptyCollection);
};
```

The `PriorityQueue` may be considered as a restricted access `KeySortedBag`. The interface is identical to that of an ordinary `Queue`, with a slightly different semantics for adding elements.

Adding elements

```
void enqueue (in any element) raises (ElementInvalid);
```

Description

Adds the element to the priority queue at a position determined by the ordering relation provided for the key type.

Exceptions

The `Element` must be the expected type; otherwise, the exception `ElementInvalid` is raised.

Removing elements

```
void dequeue () raises (EmptyCollection);
```

Description

Removes the first element from the collection.

Exceptions

The priority queue must be not be empty; otherwise, the exception `EmptyCollection` is raised.

boolean `element_dequeue` (out any element) raises (`EmptyCollection`);

Description

Retrieves the first element in the priority queue and returns it via the output parameter `element`, removes it from the priority queue, and returns the copy to the user.

Return value

Returns `true` if an element is retrieved.

Exceptions

The priority queue must not be empty; otherwise, the exception `EmptyCollection` is raised.

17.5.8 Collection Factory Interfaces

There is one collection factory defined per concrete collection interface which offers a typed operation for the creation of collection instances supporting the respective collection interface as its principal interface.

The information passed to a collection implementation at creation time is:

1. Element type specific information required to implement the correct semantics. For example, to implement `Set` semantics one has to pass the information how to test the equality of elements.
2. Element type specific information that can be exploited by the specific implementation variants. For example, a hashtable implementation of a `Set` would exploit the information how the hash value for collected elements is computed.

This element type specific information is passed to the collection implementation via an instance of a user-defined specialization of the `Operations` interface.

3. An implementation hint about the expected number of elements collected. An array based implementation may use this hint as an estimate for the initial size of the implementation array.

To enable the support for, and a user-controlled selection of implementation variants, there is a generic extensible factory defined. This allows for registration of implementation variants and their user-defined selection at creation time.

The CollectionFactory and CollectionFactories Interfaces

```
interface Operations;
interface CollectionFactory {
Collection generic_create (in ParameterList parameters) raises
(ParameterInvalid);
};
```

CollectionFactory defines a generic collection creation operation which enables extensibility and supports the creation of collection instances with the very basic capabilities.

Collection generic_create (in ParameterList parameters) raises (ParameterInvalid);

Returns a new collection instance which supports the interface **Collection** and does not offer any type checking. A sequence of name-value pairs is passed to the create operation. The only processed parameter in the given list is “expected_size,” of type “unsigned long.”

This parameter is optional and gives an estimate of the expected number of elements to be collected.

Note – All collection interface specific factories defined in this specification inherit from the interface **CollectionFactory** to enable their registration with the extensible generic **CollectionFactories** factory specified below.

```
interface CollectionFactories : CollectionFactory {
boolean add_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface, in CollectionFactory
factory);
boolean remove_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface);
};
```

The interface **CollectionFactories** specifies a generic extensible collection creation capability. It maintains a registry of collection factories. The create operation of the **CollectionFactories** does not create collection instances itself, but passes the requests through to an appropriate factory registered with it and passes the result through to the caller. Note that only factories derived from **CollectionFactory** can be registered with **CollectionFactories**.

boolean add_factory (in Istring collection_interface, in Istring impl_category, in Istring impl_interface, in CollectionFactory factory);

Registers the factory with three pieces of information:

1. **collection_interface** specifies the collection interface (directly or indirectly derived from **Collection**) supported by the given factory. That is, a collection instance created via the given factory has to support the given interface **collection_interface**.
2. **impl_interface** specifies the implementation interface (directly or indirectly derived from the interface specified in **collection_interface**) supported by the registered factory. Collection instances created via this factory are instances of this implementation interface.
3. **impl_category** specifies a named group of equivalent implementation interfaces to which the implementation interface supported by the registered factory belongs. A group of implementation interfaces of a given collection interface are equivalent if they:
 - rely on the same user-defined implementation support, that is, the same operations defined in the user-defined specialization of the **Operations** interface.
 - are based on essentially the same data structure and deliver comparable performance characteristics.

The following table lists *examples* of implementation categories (representing common implementations).

Table 17-4 Implementation Category Examples

Implementation Category	Description
ArrayBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is an array.
LinkedListBased	User-defined implementation specific operations do not have to be defined. The basic data structure used is a simple linked list.
SkipListsBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a KeyCollection derived from KeyCollection . The basic data structure are skip lists.
HashTableBased	A hash-function has to be defined for key element values that depend on whether or not the interface implemented is a KeyCollection derived from KeyCollection . The basic data structure is a hashtable based on the hash-function defined.
AVLTreeBased	A compare operation has to be defined for the key element values that depend on whether or not the collection is a KeyCollection derived from KeyCollection . The basic data structure is an AVL tree.
BStarTreeBased	A compare operation has to be defined for key values. The basic data structure is a B*tree.

The operation does not check the validity of the registration request in the sense that it checks any of the restrictions on the parameters described above, but just registers the given information with the factory. It is the responsibility of the user to ensure that the registration is valid.

The entry is added if there is not already a factory registered with the same three pieces of information; otherwise, the registration is ignored. Returns **true** if the factory is added.

boolean remove_factory (in **Istring collection_interface**, in **Istring impl_category**, in **Istring impl_interface**)

Description

Removes the factory registered with the given three pieces of information from the registry.

Return value

Returns **true** if an entry with that name exists and is removed.

create (**ParameterList parameters**) raises (**ParameterInvalid**)

The **create** operation of the **CollectionFactories** interface does not create instances itself, but passes through creation requests to factories registered with it. The factory is passed a sequence of name-value pairs of which the only mandatory one is **collection_interface** of type **Istring**.

collection_interface of type **Istring**

A string which specifies the name of the collection interface (directly or indirectly derived from **Collection**) the collection instance created has to support.

This name-value pair corresponds to the **collection_interface** parameter of the **add_factory()** operation.

The following name-value pairs are optional:

“impl_category” of type **Istring**

A string which denotes the desired implementation category. This name-value pair corresponds to the **impl_category** parameter of the **add_factory()** operation.

“impl_interface” of type **Istring**

A string which specifies a desired implementation interface. This name-value pair corresponds to the **impl_interface** parameter of the **add_factory()** operation.

If one or both of these name-value pairs are given, it is searched for a best matching entry in the factory registry and the request is passed through to the respective factory. “Best matching” means that if an implementation interface is given, it is searched for a factory supporting an exact matching implementation interface first. If no factory supporting the desired implementation interface is registered, it is searched for a factory supporting an implementation interface of the same implementation category.

If none of the two name-value pairs are given, the request is passed to a factory registered as default factory for a given “collection_interface.” For each concrete collection interface specified in this specification, there is one collection specific factory defined which serves as default factory and is assumed to be registered with `CollectionFactories`.

There must be a name-value pair with name “collection_interface” given and a factory must be registered for “collection_interface;” otherwise, the exception `ParameterInvalid` is raised.

If a desired implementation interface and/or an implementation category is given, a factory with matching characteristics must be registered; otherwise, the exception `ParameterInvalid` is raised.

For factories specified for each concrete collection interface in this specification, the following additional name-value pairs are relevant:

“operations” of type <code>Operations</code>	An instance of a user-defined specialization of <code>Operations</code> which specifies element- and/or key-type specific operations.
“expected_size” of type unsigned long	is an unsigned long and gives an estimate about the expected number of elements to be collected.

Those parameters are not processed by the create operation of `CollectionFactories` itself, but just passed through to a registered factory.

The `RACollectionFactory` and `RACollectionFactories` Interfaces

```
interface RACollectionFactory {
    RestrictedAccessCollection generic_create (in ParameterList
    parameters) raises (ParameterInvalid);
};
```

The interface `RACollectionFactory` corresponds to the interface `CollectionFactory`, but defines an abstract interface.

```
interface RACollectionFactories : RACollectionFactory {
```

```

boolean add_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface, in RACollectionFactory
factory);

boolean remove_factory (in Istring collection_interface, in Istring
impl_category, in Istring impl_interface);
};

```

The interface **RACollectionFactories** corresponds to the **CollectionFactories** interface. It enables the registration and deregistration of collections with restricted access as well as control over the implementation choice for a given restricted access collection at creation time.

The KeySetFactory Interface

```

interface KeySetFactory : CollectionFactory {
KeySet create (in Operations ops, in unsigned long expected_size);
};

```

KeySet create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **KeySet**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-5 Required element and key-type specific user-defined information for KeySetFactory. [- implied by key_compare.

KeySet						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The KeyBagFactory Interface

```

interface KeyBagFactory : CollectionFactory {
KeyBag create (in Operations ops, in unsigned long expected_size);
};

```

KeyBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **KeyBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-6 Required element and key-type specific user-defined information for KeyBagFactory. []- implied by key_compare.

KeyBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The MapFactory Interface

```
interface MapFactory : CollectionFactory {
Map create (in Operations ops, in unsigned long expected_size);
};
```

Map create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Map**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-7 Required element and key-type specific user-defined information for MapFactory. []- implied by key_compare.

Map						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The RelationFactory Interface

```
interface RelationFactory : CollectionFactory {
Relation create (in Operations ops, in unsigned long expected_size);
};
```

Relation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Relation**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-8 Required element and key-type specific user-defined information for RelationFactory.[]- implied by key_compare.

Relation						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SetFactory Interface

```
interface SetFactory : CollectionFactory {
Set create (in Operations ops, in unsigned long expected_size);
};
```

Set create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Set**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-9 Required element and key-type specific user-defined information for SetFactory.[]- implied by compare.

Set						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The BagFactory Interface

```
interface BagFactory {
Bag create (in Operations ops, in unsigned long expected_size);
};
```

Bag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Bag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-10 Required element and key-type specific user-defined information for **BagFactory.[]**- implied by **compare**.

Bag						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The KeySortedSetFactory Interface

```
interface KeySortedSetFactory {
    KeySortedSet create (in Operations ops, in unsigned long
        expected_size);
};
```

KeySortedSet create (in Operations ops, in unsigned long expected_size)

Creates and returns an instance of **KeySortedSet**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-11 Required element and key-type specific user-defined information for **KeySortedSetFactory.[]**- implied by **key_compare**.

KeySortedSet						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The KeySortedBagFactory Interface

```
interface KeySortedBagFactory : CollectionFactory {
    KeySortedBag create (in Operations ops, in unsigned long
        expected_size);
};
```

KeySortedBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **KeySortedBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-12 Required element and key-type specific user-defined information for KeySortedBagFactory.[]- implied by key_compare.

KeySortedBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

The SortedMapFactory Interface

```
interface SortedMapFactory : CollectionFactory {
SortedMap create (in Operations ops, in unsigned long
expected_size);
};
```

SortedMap create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedMap**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-13 Required element and key-type specific user-defined information for SortedMapFactory.[]- implied by key_compare.

SortedMap						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SortedRelationFactory Interface

```
interface SortedRelationFactory : CollectionFactory {
SortedRelation create (in Operations ops, in unsigned long
expected_size);
};
```

SortedRelation create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedRelation**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-14 Required element and key-type specific user-defined information for **SortedRelationFactory**.[]- implied by **key_compare**.

SortedRelation						
equal	compare	hash	key	key_equal	key_compare	key_hash
x			x	[x]	x	

The SortedSetFactory Interface

```
interface SortedSetFactory : CollectionFactory {
SortedSet create (in Operations ops, in unsigned long
expected_size);
};
```

SortedSet create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedSet**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-15 Required element and key-type specific user-defined information for **SortedSetFactory**. []- implied by **compare**.

SortedSet						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The SortedBagFactory Interface

```
interface SortedBagFactory {
SortedBag create (in Operations ops, in unsigned long
expected_size);
};
```

SortedBag create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **SortedBag**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation.

The following table defines the requirements for the element key operations to be implemented.

Table 17-16 Required element and key-type specific user-defined information for SortedBagFactory. []- implied by compare.

SortedBag						
equal	compare	hash	key	key_equal	key_compare	key_hash
[x]	x					

The SequenceFactory Interface

```
interface SequenceFactory : CollectionFactory {
Sequence create (in Operations ops, in unsigned long expected_size);
};
```

Sequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **Sequence**. No requirements on the element respectively key operations to be implemented is specified for a **Sequence**. Nevertheless one still has to pass an instance of **Operations** as type checking information has to be passed to the collection implementation.

Note – As the **Sequence** interface represents array as well as linked list implementation of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The EqualitySequence Factory Interface

```
interface EqualitySequenceFactory : CollectionFactory {
EqualitySequence create (in Operations ops, in unsigned long
expected_size);
};
```

EqualitySequence create (in Operations ops, in unsigned long expected_size);

Creates and returns an instance of **EqualitySequence**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-17 Required element and key-type specific user-defined information for **EqualitySequenceFactory**.

Equality Sequence						
equal	compare	hash	key	key_equal	key_compare	key_hash
x						

Note – As the **EqualitySequence** interface represents array as well as linked list implementations of sequentially ordered collections, a service provider should offer at least two implementations to meet the performance requirements of the two most common access patterns. That is, a service provider should offer an array based implementation and a linked list based implementation.

The HeapFactory Interface

```
interface HeapFactory : CollectionFactory {
Heap create (in Operations ops, in unsigned long expected_size);
};
```

Heap create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Heap**. No requirements for the element key operations to be implemented is specified for a **Heap**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The QueueFactory Interface

```
interface QueueFactory : RACollectionFactory {
Queue create (in Operations ops, in unsigned long expected_size);
};
```

Queue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Queue**. No requirements for the element key operations to be implemented is specified for a **Queue**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The StackFactory Interface

```
interface StackFactory : RACollectionFactory {
Stack create (in Operations ops, in unsigned long expected_size);
};
```

Stack create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Stack**. No requirements for the element key operations to be implemented is specified for a **Stack**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The DequeFactory Interface

```
interface DequeFactory : RACollectionFactory {
Deque create (in Operations ops, in unsigned long expected_size);
};
```

Deque create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **Deque**. No requirements on the element key operations to be implemented is specified for a **Deque**. Nevertheless, one still has to pass an instance of **Operations** as type checking information must pass to the collection implementation.

The PriorityQueueFactory Interface

```
interface PriorityQueueFactory : RACollectionFactory {
PriorityQueue create (in Operations ops, in unsigned long
expected_size);
};
```

PriorityQueue create (in Operations ops, in unsigned long expected_size);

Returns an instance of a **PriorityQueue**. The given instance of **Operations** passes user-defined element and key-type specific information to the collection implementation. The following table defines the requirements for the element key operations to be implemented.

Table 17-18 Required element and key-type specific user-defined information for PriorityQueueFactory. [] - implied by key_compare.

PriorityQueue						
equal	compare	hash	key	key_equal	key_compare	key_hash
			x	[x]	x	

17.5.9 Iterator Interfaces

Iterators as pointer abstraction

An **iterator** is in a first approximation of a pointer abstraction. It is a movable pointer to elements of a collection. Iterators are tightly intertwined with collections. An iterator cannot exist independently of a collection (i.e., the iterator life time cannot exceed that of the collection for which it is created). A collection is the factory for *its* iterators. An iterator is created for a given collection and can be used for this and only this collection.

The iterators specified in this specification form an interface hierarchy which parallels the collection interface hierarchy. The supported iterator movements reflect the capabilities of the corresponding collection type.

The top level **Iterator** interface defines a generic iterator usable for iteration over all types of collections. It can be set to a start position for iteration and moved via a series of forward movements through the collection visiting each element exactly once.

The **OrderedIterator** is supported by ordered collections only. It “knows about ordering;” therefore, it can be moved in forward and backward direction.

The **KeyIterator** exploits the capabilities of key collections. It can be moved to an element with a given key value, advanced to the next element with the same key value, or advanced to the next element with a different key value in iteration order.

The **KeySortedIterator** is created for key collections sorted by key. The iterator can be advanced to the previous element with the same key value or the previous element with a different key value.

The **EqualityIterator** exploits the capabilities of equality collections. It can be moved to an element with a given value, advanced to the next element with the same element value, or advanced to the next element with a different element value in iteration order.

The **EqualitySortedIterator** is created for equality collections sorted by element value. The iterator can be advanced to the previous element with the same value or the previous element with a different value.

Iterators and support for generic programming

Iterators go far beyond being simple “pointing devices.” There are essentially two reasons to extend the capabilities of iterators.

1. To support the processing of very large collections which allows for delayed instantiation or incremental query evaluation in case of very large query results. These are scenarios where the collection itself may never exist as instantiated main memory collection but is processed in “finer grains” via an iterator passed to a client.
2. To enrich the iterator with more capabilities strengthens the support for the generic programming model, as introduced with ANSI STL to the C++ world.

You can retrieve, replace, remove, and add elements via an iterator. You can test iterators for equality, compare ordered iterators, clone an iterator, assign iterators, and destroy them. Furthermore an iterator can have a `const` designation which is set when created. A `const` iterator can be used for access only.

The `reverse` iterator semantics is supported. No extra interfaces are specified to support this, but a `reverse` designation is set at creation time. An ordered iterator for which the `reverse` designation is set reinterprets the operations of a given iterator type to work in reverse.

Iterators and performance

To reduce network traffic, *combined* operations and *batch* or *bulk* operations are offered.

Combined operations are combinations of simple iterator operations often used in loops. These combinations support generic algorithms. For example, a typical combination is “test whether range end is reached; if not `retrieve_element`, advance iterator to next element.”

Batch or *bulk* operations support the retrieval, replacement, addition, and removal of many elements within one operation. In these operations, the “many elements” are always passed as a `CORBA::sequence` of elements.

The Managed Iterator Model

All iterators are managed. The real benefit of being managed is that these iterators never become undefined. Note that “undefined” is different from “invalid.” While “invalid” is a testable state and means the iterator points to nothing, “undefined” means you do not know where the iterator points to and cannot inquiry it. Changing the contents of a collection by adding or deleting elements would cause an unmanaged iterator to become “undefined.” The iterator may still point to the same element, but it may also point to another element or even “outside” the collection. As you do not know the iterator state and cannot inquiry which state the iterator has, you are forced to newly position the unmanaged iterator, for example, via a `set_to_first_element()`.

This kind of behavior, common in collection class libraries today, seems unacceptable in a distributed multi-user environment. Assume one client removes and adds elements from a collection with side effects on the unmanaged iterators of another client. The other client is not able to test whether there have been side effects on its unmanaged iterators, but would only notice them indirectly when observing strange behavior of the application.

Managed iterators are intimately related to the collection they belong to, and thus, can be informed about the changes taking place within the collection. They are always in a defined state which allows them to be used even though elements have been added or removed from the collection. An iterator may be in the state *invalid*, that is pointing to nothing. Before it can be used it has to be set to a valid position. An iterator in the

state *valid* may either point to an element (and be valid for all operations on it) or it may be in the state *in-between*, that is, not pointing to an element but still “remembering” enough state to be valid for most operations on it.

A valid managed iterator remains valid as long as the element it points to remains in the collection. As soon as the element is removed, the according managed iterator enters a so-called *in-between* state. The *in-between* state can be viewed as a vacuum within the collection. There is nothing the managed iterator can point to. Nevertheless, managed iterators remember the next (and for ordered collection, also the previous) element in iteration order. It is possible to continue using the managed iterator (in a `set_to_next_element()` for example) without resetting it first.

There are some limitations. Once a managed iterator no longer points to an element, it remembers the iteration order in which the element stood before it was deleted. However, it does not remember the element itself. Thus, there are some operations which cannot be performed even though a managed iterator is used.

Consider an iteration over a **Bag**, for example. If you iterate over all different elements with the iterator operation `set_to_next_different_element()`, then removing the element the iterator points to leads to an undefined behavior of the collection later on. By removing the element, the iterator becomes *in-between*. The `set_to_next_different_element()` operation then has no chance to find the next different element as the collection does not know what is different in terms of the current iterator state. Likewise, for a managed iterator in the state *in-between* all operations ending with “...at” are not defined. The reason is simple: There is no element at the iterator’s position - nothing to retrieve, to replace, or to remove in it. This situation is handled by raising an exception `IteratorInvalid`.

Additionally, all operations that (potentially) destroy the iteration order of a collection invalidate the corresponding managed iterators that have been in the state *in-between* before the operation was invoked. These are the `sort()` and the `reverse()` operation.

The Iterator Interface

```
// Iterators

interface Iterator {

    // moving iterators
    boolean set_to_first_element ();
    boolean set_to_next_element() raises (IteratorInvalid);
    boolean set_to_next_nth_element (in unsigned long n) raises
    (IteratorInvalid);

    // retrieving elements
    boolean retrieve_element (out any element) raises (IteratorInvalid,
    IteratorInBetween);
```

```

boolean retrieve_element_set_to_next (out any element, out boolean
more) raises (IteratorInvalid, IteratorInBetween);

boolean retrieve_next_n_elements (in unsigned long n, out
AnySequence result, out boolean more) raises (IteratorInvalid,
IteratorInBetween);

boolean not_equal_retrieve_element_set_to_next (in Iterator test,
out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements
void remove_element() raises (IteratorInvalid, IteratorInBetween);
boolean remove_element_set_to_next() raises (IteratorInvalid,
IteratorInBetween);

boolean remove_next_n_elements (in unsigned long n, out unsigned
long actual_number) raises (IteratorInvalid, IteratorInBetween);
boolean not_equal_remove_element_set_to_next (in Iterator test)
raises (IteratorInvalid, IteratorInBetween);

// replacing elements
void replace_element (in any element) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean replace_element_set_to_next (in any element)
raises (IteratorInvalid, IteratorInBetween, ElementInvalid);
boolean replace_next_n_elements (in AnySequence elements, out
unsigned long actual_number) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean not_equal_replace_element_set_to_next (in Iterator test, in
any element) raises (IteratorInvalid, IteratorInBetween,
ElementInvalid);

// adding elements
boolean add_element_set_iterator (in any element) raises
(ElementInvalid);
boolean add_n_elements_set_iterator (in AnySequence elements, out
unsigned long actual_number) raises (ElementInvalid);

// setting iterator state
void invalidate ();
// testing iterators
boolean is_valid ();
boolean is_in_between ();
boolean is_for (in Collection collector);
boolean is_const ();
boolean is_equal (in Iterator test) raises (IteratorInvalid);

// cloning, assigning, destroying an iterators

```

```
Iterator clone ();  
void assign (in Iterator from_where) raises (IteratorInvalid);  
void destroy ();  
};
```

Moving iterators

```
boolean set_to_first_element ();
```

Description

The iterator is set to the first element in iteration order of the collection it belongs to. If the collection is empty, that is, if no first element exists, the iterator is invalidated.

Return value

Returns **true** if the collection it belongs to is not empty.

```
boolean set_to_next_element () raises (IteratorInvalid);
```

Description

Sets the iterator to the next element in the collection in iteration order or invalidates the iterator if no more elements are to be visited. If the iterator is in the state *in-between*, the iterator is set to its “potential next” element.

Return value

Returns **true** if there is a next element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

```
boolean set_to_next_nth_element (in unsigned long n) raises (IteratorInvalid);
```

Description

Sets the iterator to the element *n* movements away in collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between* the movement to the “potential next” element is the first of the *n* movements.

Return value

Returns **true** if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

Retrieving elements

boolean `retrieve_element` (out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed and returns it via the output parameter `element`.

Return value

Returns `true` if an element was retrieved.

Exceptions

The iterator must point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Note – Whether a copy of the element is returned or the element itself depends on the element type represented by the `any`. If it is an object, a reference to the object in the collection is returned. If the element type is a non-object type, a copy of the element is returned. In case of element type object, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

boolean `retrieve_element_set_to_next` (out any element) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed to and returns it via the output parameter `element`. The iterator is moved to the next element in iteration order. If there is a next element `more` is set to `true`. If there are no more next elements, the iterator is invalidated and `more` is set to `false`.

Return value

Returns `true` if an element was retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

boolean `retrieve_next_n_elements` (in unsigned long `n`, out AnySequence `result`, out boolean `more`) raises (IteratorInvalid, IteratorInBetween);

Description

Retrieves at most the next `n` elements in iteration order of the iterator's collection and returns them as **sequence** of anys via the output parameter **result**. Counting starts with the element the iterator points to. The iterator is moved behind the last element retrieved. If there is an element behind the last element retrieved, **more** is set to **true**. If there are no more elements behind the last element retrieved or there are less than `n` elements for retrieval, the iterator is invalidated and **more** is set to **false**. If the value of `n` is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns **true** if at least one element is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

boolean `not_equal_retrieve_element_set_to_next` (in Iterator `test`, out any element) raises (IteratorInvalid, IteratorInBetween);

Description

Compares the given iterator **test** with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is moved to the next element, and **true** is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is not moved to the next element, and **false** is returned.

Return value

Returns **true** if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator **test** each must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

Removing elements

void `remove_element` () raises (IteratorInvalid, IteratorInBetween);

Description

Removes the element pointed to by this iterator and sets the iterator *in-between*.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

`boolean remove_element_set_to_next()` (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes the element pointed to by this iterator and moves the iterator to the next element.

Return value

Returns `true` if a next element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

`boolean remove_next_n_elements` (in unsigned long `n`, out unsigned long `actual_number`) raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes at most the next `n` elements in iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the next element behind the last element removed. If there are no more elements behind the last element removed or there are less than `n` elements for removal, the iterator

is invalidated. If the value of `n` is 0, all elements in the collection are removed until the end is reached. The output parameter `actual_number` is set to the actual number of elements removed. If the value of `n` is 0, all elements in the collection are removed until the end is reached.

Return value

Returns **true** if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have the `const` designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

boolean `not_equal_remove_element_set_to_next(in iterator test)`
(`IteratorInvalid`, `IteratorInBetween`);

Description

Compares this iterator with the given iterator `test`. If they are not equal the element this iterators points to is removed and the iterator is set to the next element, and **true** is returned. If they are equal the element pointed to is removed, the iterator is set *in-between*, and **false** is returned.

Return value

Returns **true** if this iterator and the given iterator `test` are not equal when the operations starts.

Exception

This iterator and the given iterator `test` must be valid otherwise the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator and the given iterator `test` must not have a `const` designation otherwise the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

Replacing elements

void `replace_element` (in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces the element pointed to by the given element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

boolean `replace_element_set_to_next`(in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the next element. If there are no more elements, the iterator is invalidated.

Return value

Returns `true` if there is a next element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

`boolean replace_next_n_elements(in AnySequence elements, out unsigned long actual_number)` raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Replaces at most as many elements in iteration order as given in `elements` by the given elements. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter `actual_number`.

The iterator is moved to the next element behind the last element replaced. If there are no more elements behind the last element replaced or the number of elements in the collection to be replaced is less than the number given `elements`, the iterator is invalidated.

Return value

Returns `true` if there is another element behind the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The elements given must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception `ElementInvalid` is raised.

For positioning property see “The Collection Interface” on page 17-21.

`boolean not_equal_replace_element_set_to_next` (in `Iterator` test, in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Compares this iterator and the given iterator `test`. If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the next element, and `true` is returned. If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the next element, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are not equal before the operations starts.

Exceptions

This iterator and the given iterator must be valid and point to an element each; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

Adding elements

boolean `add_element_set_iterator` (in any element) (`ElementInvalid`);

Description

Adds an element to the collection that this iterator points to and sets the iterator to the added element. The exact semantics depends on the properties of the collection for which this iterator is created.

If the collection supports unique elements or keys and the element or key is already contained in the collection, adding is ignored and the iterator is just set to the element or key already contained. In sequential collections, the element is always added as last element. In sorted collections, the element is added at a position determined by the element or key value.

Return value

Returns `true` if the element was added. The element to be added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Exceptions

If the collection is a Map and contains an element with the same key as the given element, then this element has to be equal to the given element; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

void `add_n_elements_set_iterator` (in AnySequence elements, out unsigned long actual_number) (`ElementInvalid`);

Description

Adds the given elements to the collection that this iterator points to. The elements are added in the order of the input sequence of elements and the delivered semantics is consistent with the semantics of the `add_element_set_iterator` operation. It is essentially a sequence of `add_element_set_iterator` operations. The output parameter `actual_number` is set to the number of elements added.

Setting iterator state

```
void invalidate ();
```

Description

Sets the iterator to the state *invalid*, that is, “pointing to nothing.” You may also say that the iterator, in some sense, is set to “NULL.”

Testing iterators

Whenever there is a precondition for an iterator operation to be checked, there is a test operation provided that enables the user to avoid raising an exception.

```
boolean is_valid ();
```

Return value

Returns **true** if the Iterator is *valid*, that is points to an element of the collection or is in the state *in-between*.

```
boolean is_for (in Collection collector);
```

Return value

Returns **true** if this iterator can operate on the given collection.

```
boolean is_const ();
```

Return value

Returns **true** if this iterator is created with “const” designation.

```
boolean is_in_between ();
```

Return value

Returns **true** if the iterator is in the state *in-between*.

boolean is_equal (in Iterator test) raises (IteratorInvalid);

Return value

Returns **true** if the given iterator points to the identical element as this iterator.

Exceptions

The given iterator must belong to the same collection as the iterator; otherwise, the exception `IteratorInvalid` is raised.

Cloning, Assigning, Destroying iterators

Iterator clone();

Description

Creates a copy of this iterator.

void assign (in Iterator from_where) raises (IteratorInvalid)

Description

Assigns the given iterator to this iterator.

Exceptions

The given iterator must be created for the same collection as this iterator; otherwise, the exception `IteratorInvalid` is raised.

void destroy();

Description

Destroys this iterator.

The OrderedIterator Interface

```
interface OrderedIterator: Iterator {
```

```
    // moving iterators
```

```
    boolean set_to_last_element ();
```

```
    boolean set_to_previous_element() raises (IteratorInvalid);
```

```
    boolean set_to_nth_previous_element(in unsigned long n) raises  
    (IteratorInvalid);
```

```
void set_to_position (in unsigned long position) raises
(PositionInvalid);

// computing iterator position
unsigned long position () raises (IteratorInvalid);

// retrieving elements
boolean retrieve_element_set_to_previous(out any element, out
boolean more) raises (IteratorInvalid, IteratorInBetween);
boolean retrieve_previous_n_elements (in unsigned long n, out
AnySequence result, out boolean more) raises (IteratorInvalid,
IteratorInBetween);
boolean not_equal_retrieve_element_set_to_previous (in Iterator
test, out any element) raises (IteratorInvalid, IteratorInBetween);

// removing elements
boolean remove_element_set_to_previous() raises (IteratorInvalid,
IteratorInBetween);
boolean remove_previous_n_elements (in unsigned long n, out unsigned
long actual_number) raises (IteratorInvalid, IteratorInBetween);
boolean not_equal_remove_element_set_to_previous(in Iterator test)
raises (IteratorInvalid, IteratorInBetween);

// replacing elements
boolean replace_element_set_to_previous(in any element) raises
(IteratorInvalid, IteratorInBetween, ElementInvalid);
boolean replace_previous_n_elements(in AnySequence elements, out
unsigned long actual_number) raises (IteratorInvalid,
IteratorInBetween, ElementInvalid);
boolean not_equal_replace_element_set_to_previous (in Iterator
test, in any element) raises (IteratorInvalid,IteratorInBetween,
ElementInvalid);

// testing iterators
boolean is_first ();
boolean is_last ();
boolean is_for_same (in Iterator test);
boolean is_reverse ();
};
```

Moving iterators

```
boolean set_to_last_element();
```

Description

Sets the iterator to the last element of the collection in iteration order. If the collection is empty (if no last element exists) the given iterator is invalidated.

Return value

Returns **true** if the collection is not empty.

`boolean set_to_previous_element()` raises (IteratorInvalid);

Description

Sets the iterator to the previous element in iteration order, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the iterator is set to its “potential previous” element.

Return value

Returns **true** if a previous element exists.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

`boolean set_to_nth_previous_element (in unsigned long n)` raises (IteratorInvalid);

Description

Sets the iterator to the element *n* movements away in reverse collection iteration order or invalidates the iterator if there is no such element. If the iterator is in the state *in-between*, the movement to the “potential previous” element is the first of the *n* movements.

Return value

Returns **true** if there is such an element.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

`void set_to_position (in unsigned long position)` raises (PositionInvalid);

Description

Sets the iterator to the element at the given position. Position 1 specifies the first element.

Exceptions

Position must be a valid position (i.e., greater than or equal to 1 and less than or equal to `number_of_elements()`); otherwise, the exception `PositionInvalid` is raised.

Computing iterator position

`unsigned long position ()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Determines and returns the current position of the iterator. Position 1 specifies the first element.

Exceptions

The iterator must be pointing to an element of the collection; otherwise, the exception `IteratorInvalid` respectively `IteratorInBetween` is raised.

Retrieving elements

`boolean retrieve_element_set_to_previous (out any element, out boolean more)` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves the element pointed to and returns it via the output parameter `element`. The iterator is set to the previous element in iteration order. If there is a previous element, `more` is set to `true`. If there are no more previous elements, the iterator is invalidated and `more` is set to `false`.

Return value

Returns `true` if an element was returned.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

`boolean retrieve_previous_n_elements(in unsigned long n, out AnySequence result, out boolean more)` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Retrieves at most the *n* previous elements in iteration order of this iterator's collection and returns them as **sequence** of **anys** via the output parameter **result**. Counting starts with the element the iterator is pointing to. The iterator is moved to the element before the last element retrieved.

- If there is an element before the last element retrieved, **more** is set to **true**.
- If there are no more elements before the last element retrieved or there are less than *n* elements for retrieval, the iterator is invalidated and **more** is set to **false**.
- If the value of *n* is 0, all elements in the collection are retrieved until the end is reached.

Return value

Returns **true** if at least one element is retrieved.

Exceptions

The iterator must be valid and pointing to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

boolean not_equal_retrieve_element_set_to_previous (in **Iterator test**, out any element) raises (**IteratorInvalid**, **IteratorInBetween**);

Description

Compares the given iterator **test** with this iterator.

- If they are not equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is moved to the previous element, and **true** is returned.
- If they are equal, the element pointed to by this iterator is retrieved and returned via the output parameter **element**, the iterator is not moved to the previous element, and **false** is returned.

Return value

Returns **true** if this iterator is not equal to the test iterator at the beginning of the operation.

Exceptions

The iterator and the given iterator **test** each must be valid and point to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

Replacing elements

boolean replace_element_set_to_previous(in any element) raises (**IteratorInvalid**, **IteratorInBetween**, **ElementInvalid**);

Description

Replaces the element pointed to by this iterator by the given element and sets the iterator to the previous element. If there are no previous elements, the iterator is invalidated.

Return value

Returns **true** if there is a previous element.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

The iterator must not have a **const** designation; otherwise, the exception **IteratorInvalid** is raised.

The element must be the expected element type; otherwise, the **ElementInvalid** exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception **ElementInvalid** is raised.

For positioning properties, see “The Collection Interface” on page 17-21.

boolean `replace_previous_n_elements`(in **AnySequence** `elements`, out **unsigned long** `actual_number`) raises (**IteratorInvalid**, **IteratorInBetween**, **ElementInvalid**);

Description

At most, replaces as many elements in reverse iteration order as given in **elements**. Counting starts with the element the iterator points to. If there are less elements in the collection left to be replaced than the given number of elements as many elements as possible are replaced and the actual number of elements replaced is returned via the output parameter `actual_number`.

The iterator is moved to the element before the last element replaced. If there are no more elements before the last element replaced or the number of elements in the collection to be replaced is less than the number of given elements, the iterator is invalidated.

Return value

Returns **true** if there is an element before the last element replaced.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

The elements given must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

For each element the positioning property of the replaced element must be the same as that of the element replacing it; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

boolean `not_equal_replace_element_set_to_previous` (in `Iterator` test, in any element) raises (`IteratorInvalid`, `IteratorInBetween`, `ElementInvalid`);

Description

Compares this iterator and the given iterator `test`.

- If they are not equal, the element pointed to by this iterator is replaced by the given element, the iterator is set to the previous element, and `true` is returned.
- If they are equal, the element pointed to by this iterator is replaced by the given element, the iterator is not set to the previous element, and `false` is returned.

Return value

Returns `true` if this iterator and the given iterator `test` are not equal before the operations starts.

Exceptions

This iterator and the given iterator each must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator must not have a `const` designation; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected element type; otherwise, the `ElementInvalid` exception is raised.

The given element must have the same positioning property as the replaced element; otherwise, the exception `ElementInvalid` is raised.

For positioning property, see “The Collection Interface” on page 17-21.

Removing elements

boolean `remove_element_set_to_previous()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Removes the element pointed to by this iterator and moves the iterator to the previous element.

Return value

Returns **true** if a previous element exists.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception **IteratorInvalid** is raised.

The iterator must not have the **const** designation; otherwise, the exception **IteratorInvalid** is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

boolean **remove_previous_n_elements** (in unsigned long **n**, out unsigned long **actual_number**) raises (**IteratorInvalid**, **IteratorInBetween**);

Description

Removes at most the previous **n** elements in reverse iteration order of the iterator's collection. Counting starts with the element the iterator points to. The iterator is moved to the element before the last element removed.

- If there are no more elements before the last element removed or there are less than **n** elements for removal, the iterator is invalidated.
- If the value of **n** is 0, all elements in the collection are removed until the beginning is reached. The output parameter **actual_number** is set to the actual number of elements removed.

Return value

Returns **true** if the iterator is not invalidated.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

The iterator must not have the **const** designation; otherwise, the exception **IteratorInvalid** is raised.

Side effects

Other valid iterators pointing to removed elements go *in-between*.

All other iterators keep their state.

boolean **not_equal_remove_element_set_to_previous**(in **Iterator** **test**) raises (**IteratorInvalid**, **IteratorInBetween**);

Description

Compares this iterator with the given iterator **test**.

- If they are not equal, the element this iterator points to is removed, the iterator is set to the previous element, and **true** is returned.
- If they are equal, the element pointed to is removed, the iterator is set *in-between*, and **false** is returned.

Return value

Returns **true** if this iterator and the given iterator **test** are equal when the operation starts.

Exceptions

This iterator and the given iterator **test** must be valid; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

This iterator and the given iterator **test** must not have a **const** designation; otherwise, the exception `IteratorInvalid` is raised.

Side effects

Other valid iterators pointing to the removed element go *in-between*.

All other iterators keep their state.

Testing iterators

`boolean is_first ();`

Return value

Returns **true** if the iterator points to the first element of the collection it belongs to.

`boolean is_last ();`

Return value

Returns **true** if the iterator points to the last element of the collection it belongs to.

`boolean is_for_same (in Iterator test);`

Return value

Returns **true** if the given iterator is for the same collection as this.

`boolean is_reverse();`

Return value

Returns **true** if the iterator is created with “reverse” designation.

The SequentialIterator Interface

```
interface SequentialIterator : OrderedIterator {  
    // adding elements  
    boolean add_element_as_next_set_iterator (in any element)  
    raises(IteratorInvalid, ElementInvalid);  
    void add_n_elements_as_next_set_iterator(in AnySequence elements)  
    raises(IteratorInvalid, ElementInvalid);  
  
    boolean add_element_as_previous_set_iterator(in any element)  
    raises(IteratorInvalid, ElementInvalid);  
    void add_n_elements_as_previous_set_iterator(in AnySequence  
    elements) raises(IteratorInvalid, ElementInvalid);  
};
```

Adding elements

```
boolean add_element_as_next_set_iterator (in any element)  
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the element to the collection that this iterator points to (in iteration order) behind the element this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added before the “potential next” element.

Return value

Returns **true** if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

```
void add_n_elements_as_next_set_iterator(in AnySequence elements)  
raises(IteratorInvalid, ElementInvalid);
```

Description

Adds the given elements to the collection that this iterator points to behind the element the iterator points to. The behavior is the same as *n* times calling the operation `add_element_as_next_set_iterator()`.

If the iterator is in the state *in-between*, the elements are added before the “potential next” element.

The elements are added in the order given in the input sequence.

`boolean add_element_as_previous_set_iterator(in any element)`
`raises(IteratorInvalid, ElementInvalid)`

Description

Adds the element to the collection that this iterator points to (in iteration order) before the element that this iterator points to and sets the iterator to the element added. If the iterator is in the state *in-between*, the element is added after the “potential previous” element.

Return value

Returns `true` if the element is added.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element added must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

Side effects

All other iterators keep their state.

`void add_n_elements_as_previous_set_iterator(in AnySequence elements)`
`raises(IteratorInvalid, ElementInvalid);`

Description

Adds the given elements to the collection that this iterator points to previous to the element the iterator points to. The behavior is the same as *n* times calling the operation `add_element_as_previous_set_to_next()`.

If the iterator is in the state *in-between*, the elements are added behind the “potential previous” element.

The elements are added in the reverse order given in the input sequence.

The KeyIterator Interface

```
interface KeyIterator : Iterator {  
    // moving the iterators  
    boolean set_to_element_with_key (in any key) raises(KeyInvalid);  
    boolean set_to_next_element_with_key (in any key)  
    raises(IteratorInvalid, KeyInvalid);  
    boolean set_to_next_element_with_different_key() raises  
    (IteratorInBetween, IteratorInvalid);  
  
    // retrieving the keys  
    boolean retrieve_key (out any key) raises (IteratorInBetween,  
    IteratorInvalid);  
    boolean retrieve_next_n_keys (out AnySequence keys) raises  
    (IteratorInBetween, IteratorInvalid);  
};
```

Moving iterators

boolean set_to_element_with_key (in any key) raises (KeyInvalid);

Description

Locates an element in the collection with the same key as the given key. Sets the iterator to the element located or invalidates the iterator if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception **KeyInvalid** is raised.

boolean set_to_next_element_with_key (in any key) raises (IteratorInvalid, KeyInvalid);

Description

Locates the next element in iteration order with the same key value as the given key, starting search at the element next to the one pointed to by the iterator. Sets the iterator to the element located.

- If there is no such element, the iterator is invalidated.

- If the iterator is in the state *in-between*, locating starts at the iterator's "potential next" element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_next_element_with_different_key ()` raises (`IteratorInBetween`, `IteratorInvalid`)

Description

Locates the next element in iteration order with a key different from the key of the element pointed to by the iterator, starting the search with the element next to the one pointed to by the iterator. Sets the iterator to the located element.

If no such element exists, the iterator is invalidated.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` respectively `IteratorInvalid` is raised.

Retrieving keys

`boolean key (out any key)` raises(`IteratorInvalid`,`IteratorInBetween`);

Description

Retrieves the key of the element this iterator points to and returns it via the output parameter **key**.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

`boolean retrieve_next_n_keys (in unsigned long n, out AnySequence keys)`
raises(`IteratorInvalid`, `IteratorInbetween`)

Description

Retrieves the keys of at most the next *n* elements in iteration order, sets the iterators to the element behind the last element from which a key is retrieved, and returns them via the output parameter **keys**. Counting starts with the element this iterator points to.

- If there is no element behind the last element from which a key is retrieved or there are less than *n* elements to retrieve keys from the iterator is invalidated.
- If the value of *n* is 0, the keys of all elements in the collection are retrieved until the end is reached.

Return value

Returns **true** if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception **IteratorInvalid** or **IteratorInBetween** is raised.

The EqualityIterator Interface

```
interface EqualityIterator : Iterator {  
    // moving the iterators  
    boolean set_to_element_with_value(in any element)  
    raises(ElementInvalid);  
    boolean set_to_next_element_with_value(in any element)  
    raises(IteratorInvalid, ElementInvalid);  
    boolean set_to_next_element_with_different_value() raises  
    (IteratorInBetween, IteratorInvalid);  
};
```

Moving iterators

boolean set_to_element_with_value (in any element) raises(**ElementInvalid**);

Description

Locates an element in the collection that is equal to the given element. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the collection contains several such elements, the first element in iteration order is located.

Return value

Returns **true** if an element is found.

Exceptions

The element must be of the expected type; otherwise, the expected `ElementInvalid` is raised.

`boolean set_to_next_element_with_value(in any element)` raises (`IteratorInvalid`, `ElementInvalid`);

Description

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the iterator. Sets the iterator to the located element in the collection.

- If there is no such element, the iterator is invalidated.
- If the iterator is in the state *in-between*, locating is started at the iterator's "potential next" element.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_next_different_element ()` raises (`IteratorInvalid`, `IteratorInBetween`);

Description

Locates the next element in iteration order that is different from the element pointed to. Sets the iterator to the located element, or if no such element exists, the iterator is invalidated.

Return value

Returns `true` if the next different element was found.

Exceptions

The iterator must be valid and point to an element of the collection; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The EqualityKeyIterator Interface

```
interface EqualityKeyIterator : EqualityIterator, KeyIterator {};
```

This interface just combines the two interfaces `EqualityIterator` (see “The EqualityIterator Interface” on page 17-110) and `KeyIterator` (see “The KeyIterator Interface” on page 17-108).

The SortedIterator Interface

```
interface SortedIterator : OrderedIterator {};
```

This interface does not add any new operations but new semantics to the operations.

The KeySortedIterator Interface

```
// enumeration type for specifying ranges
enum LowerBoundStyle {equal_lo, greater, greater_or_equal};
enum UpperBoundStyle {equal_up, less, less_or_equal};
interface KeySortedIterator : KeyIterator, SortedIterator
{
    // moving the iterators
    boolean set_to_first_element_with_key (in any key, in
    LowerBoundStyle style) raises(KeyInvalid);
    boolean set_to_last_element_with_key (in any key, in UpperBoundStyle
    style) raises (KeyInvalid);
    boolean set_to_previous_element_with_key (in any key)
    raises(IteratorInvalid, KeyInvalid);
    boolean set_to_previous_element_with_different_key() raises
    (IteratorInBetween, IteratorInvalid);
    // retrieving keys
    boolean retrieve_previous_n_keys(out AnySequence keys) raises
    (IteratorInBetween, IteratorInvalid);
};
```

Moving iterators

```
boolean set_to_first_element_with_key (in any key, in LowerBoundStyle style)
raises (KeyInvalid);
```

Description

Locates the first element in iteration order in the collection with key:

- equal to the given key, if `style` is `equal_lo`
- greater or equal to the given key, if `style` is `greater_or_equal`
- greater than the given key, if `style` is `greater`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_last_element_with_key(in any key, in UpperBoundStyle style);`

Description

Locates the last element in iteration order in the collection with key:

- equal to the given key, if `style` is `equal_up`
- less or equal to the given key, if `style` is `less_or_equal`
- less than the given key, if `style` is `less`

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_previous_element_with_key (in any key) raises(IteratorInvalid, KeyInvalid);`

Description

Locates the previous element in iteration order with a key equal to the given key, beginning at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The key must be of the expected type; otherwise, the exception `KeyInvalid` is raised.

`boolean set_to_previous_element_with_different_key()` raises
(`IteratorInBetween`, `IteratorInvalid`);

Description

Locates the previous element in iteration order with a key different from the key of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns `true` if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

Retrieving keys

`boolean retrieve_previous_n_keys` (in unsigned long `n`, out `AnySequence` `keys`)
raises(`IteratorInvalid`, `IteratorInbetween`)

Description

Retrieves the keys of at most the previous `n` elements in iteration order, sets the iterators to the element before the last element from which a key is retrieved, and returns them via the output parameter `keys`. Counting starts with the element this iterator points to.

- If there is no element previous the one from which the `n`th key is retrieved or if there are less than `n` elements to retrieve keys from, the iterator is invalidated.
- If the value of `n` is 0, the keys of all elements in the collection are retrieved until the beginning is reached.

Return value

Returns `true` if at least one key is retrieved.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInvalid` or `IteratorInBetween` is raised.

The EqualitySortedIterator Interface

```
interface EqualitySortedIterator : EqualityIterator, SortedIterator
{
    // moving the iterator
```

```

boolean set_to_first_element_with_value (in any element, in
LowerBoundStyle style) raises (ElementInvalid);

boolean set_to_last_element_with_value (in any element, in
UpperBoundStyle style) raises (ElementInvalid);

boolean set_to_previous_element_with_value (in any elementally)
raises (IteratorInvalid, ElementInvalid);

boolean set_to_previous_element_with_different_value() raises
(IteratorInBetween, IteratorInvalid);
};

```

Moving iterators

boolean set_to_first_element_with_value (in any element, in LowerBoundStyle style) raises(ElementInvalid);

Description

Locates the first element in iteration order in the collection with value:

- equal to the given element value, if style is **equal_lo**
- greater or equal to the given element value, if style is **greater_or_equal**
- greater than the given element value, if style is **greater**

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

boolean set_to_last_element_with_value(in any element, in UpperBoundStyle style) raises (ElementInvalid);

Description

Locates the last element in iteration order in the collection with value:

- equal to the given element value, if style is **equal_up**
- less or equal to the given element value, if style is **less_or_equal**
- less than the given element value, if style is **less**

Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_previous_element_with_value(in any element)
raises(IteratorInvalid, ElementInvalid);`

Description

Locates the previous element in iteration order with a value equal to the given element value, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, the search begins at the iterator's "potential previous" element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

`boolean set_to_previous_element_with_different_value() raises
(IteratorInBetween, IteratorInvalid);`

Description

Locates the previous element in iteration order with a value different from the value of the element pointed to, beginning search at the element previous to the one pointed to and moving in reverse iteration order through the elements. Sets the iterator to the located element, or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid and point to an element; otherwise, the exception `IteratorInBetween` or `IteratorInvalid` is raised.

The EqualityKeySortedIterator Interface

```
interface EqualityKeySortedIterator: EqualitySortedIterator,
KeySortedIterator {};
```

This interface combines the interfaces **KeySortedIterator** and **EqualitySortedIterator**. This interface does not add any new operations, but new semantics.

The EqualitySequentialIterator Interface

```
interface EqualitySequentialIterator : EqualityIterator,
SequentialIterator
{
// locating elements
boolean set_to_first_element_with_value (in any element) raises
(ElementInvalid);
boolean set_to_last_element_with_value (in any element) raises
(ElementInvalid);
boolean set_to_previous_element_with_value (in any element) raises
(ElementInvalid);
};
```

Moving Iterators

```
boolean set_to__first_element_with_value (in any element)
raises(ElementInvalid);
```

Description

Sets the iterator to the first element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception **ElementInvalid** is raised.

```
boolean set_to_last_element (in any element) raises(ElementInvalid);
```

Description

Sets the iterator to the last element in iteration order in the collection that is equal to the given element or invalidates the iterator if no such element exists.

Return value

Returns **true** if an element was found.

Exceptions

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

boolean `set_to_previous_element_with_value` (in any element) raises (`IteratorInvalid`, `ElementInvalid`);

Description

Sets the iterator to the previous element in iteration order that is equal to the given element, beginning search at the element previous to the one specified by the iterator and moving in reverse iteration order through the elements. Sets the iterator to the located element or invalidates the iterator if no such element exists. If the iterator is in the state *in-between*, search starts at the “potential previous” element.

Return value

Returns **true** if an element was found.

Exceptions

The iterator must be valid; otherwise, the exception `IteratorInvalid` is raised.

The element must be of the expected type; otherwise, the exception `ElementInvalid` is raised.

17.5.10 Function Interfaces

The Operations Interface

```
Interface Operations {

    // element type specific information
    readonly attribute CORBA::TypeCode element_type;
    boolean check_element_type (in any element);
    boolean equal (in any element1, in any element2);
    long compare (in any element1, in any element2);
    unsigned long hash (in any element, in unsigned long value);

    // key retrieval
    any key (in any element);

    // key type specific information
```

```

readonly attribute CORBA::TypeCode key_type;
boolean check_key_type (in any key);
boolean key_equal (in any key1, in any key2);
long key_compare (in any key1, in any key2);
unsigned long key_hash (in any thisKey, in unsigned long value);

// destroying
void destroy();
};

```

The function interface **Operations** is used to pass a number of other user-defined element type specific information to the collection implementation.

The first kind of element type specific information passed is used for typechecking. There are attributes specifying the element and key type expected in a given collection. In addition to the type information there are two typechecking operations which allow customizing the typechecking in a user-defined manner. The “default semantics” of these operations is a simple check on whether the type code of the given element or key exactly matches the type code specified in the element key type attribute.

Dependent on the properties as represented by a collection interface the respective implementation relies on some element type specific or key type specific information to be passed to it. For example one has to pass the information “element comparison” to implementation of a **SortedSet** or “key equality” to the implementation of a **KeySet** to guarantee uniqueness of keys. To pass this information, the **Operations** interface is used.

The third use of this interface is to pass element or key type specific information relevant for different categories of implementations. (Performing) implementations of associative collections essentially can be partitioned into the categories comparison-based or hashing-based. An AVL-tree implementation for a **KeySet** (for example) is key-comparison-based; therefore, it relies on key comparison defined and a hash table implementation of **KeySet** hashing-based (which relies on the information how a hash key values). Passing this information is the third kind of usage of the **Operations** interface.

The operations defined in the **Operations** interface are in summary:

- element type checking and key type checking
- element equality and the ordering relationship on elements
- key equality and ordering relationship on keys
- key access
- hash information on elements and keys

In order to pass this information to the collection, a user has to derive and implement an interface from the interface **Operations**. Which operations you have to implement depends on the collection interface and the implementation category you want to use. An instance of this interface is passed to a collection at creation time and then can be used by the implementation.

Ownership for an **Operations** instance is passed to the collection at creation time. That is, the same instance of **Operations** respectively a derived interface cannot be used in another collection instance. The collection is responsible for destroying the **Operations** instance when the collection is destroyed.

Operations only defines an abstract interface. Specialization and implementation are part of the application development as is the definition and implementation of respective factories and are not listed in this specification.

Element type specific operations

readonly attribute CORBA::TypeCode element_type;

Description

Specifies the type of the element to be collected.

boolean check_element_type (in any element);

Description

A collection implementation may rely on this operation being defined to use it for its type checking. A default implementation may be a simple test whether the type code of the given element exactly matches **element_type**. For object references, sometimes a check on equality of the type codes is not desired but a check on whether the type of the given element is a specialization of the **element_type**.

Return value

Returns **true** if the given element passed the user-defined element type-checking.

boolean equal (in any element1, in any element2);

Return value

Returns **true** if **element1** is equal to **element2** with respect to the user-defined semantics of element equality.

Note – If case **compare** is defined, the equal operation has to be consistently defined (i.e., is implied by the defined element comparison).

long compare (in any element1, in any element2);

Return value

Returns a value less than zero if `element1 < element2`, zero if the values are equal, and a value greater than zero if `element1 > element2` with respect to the user-defined ordering relationship on elements.

unsigned long hash (in any element, in unsigned long value);

Return value

Returns a user-defined hash value for the given `element`. The given `value` specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than `value`.

Note – The definition of the hash function has to be consistent with the defined element equality (i.e., if two elements are equal with respect to the user-defined element equality they have to be hashed to the same hash value).

Computing the key

any key (in any element);

Description

Computes the (user-defined) key of the given element.

Key type specific information

readonly attribute CORBA::TypeCode key_type;

Description

Specifies the type of the key of the elements to be collected.

boolean check_key_type (in any key);

Return value

Returns `true` if the given key passed the user-defined element type-checking.

boolean key_equal (in any key1, in any key2);

Return value

Returns **true** if **key1** is equal to **key2** with respect to the user-defined semantics of key equality.

Note – If case **key_compare** is defined, the **key_equal** operation has to be consistently defined (i.e., is implied by the defined key comparison). When both key and element equality are defined, the definitions have to be consistent in the sense that element equality has to imply key equality.

key_compare (in any **key1**, in any **key2**);

Return value

Returns a value less than zero if **key1** < **key2**, zero if the values are equal, and a value greater than zero if **key1** > **key2** with respect to the user-defined ordering relationship on keys.

unsigned long key_hash (in any **key**, in **unsigned long value**);

Return value

Returns a user defined hash value for the given **key**. The given **value** specifies the size of the hashtable. This information can be used for the implementation of more or less sophisticated hash functions. Computed hash values have to be less than **value**.

Note – The definition of the hash function has to be consistent with the defined key equality (i.e., if two elements are equal with respect to the user defined element equality they have to be hashed to the same hash value).

Destroying the Operations instance

void destroy();

Destroys the operations instance.

The Command and Comparator Interface

Command and Comparator are auxiliary interfaces.

A collection service provider may either provide the interfaces only or a default implementation that raises an exception whenever an operation of these interfaces is called. In either case, a user is forced to provide his/her implementation of either the interfaces or a derived interface to make use of them in the operations `all_elements_do`, and `sort`.

The Command Interface

An instance of an interface derived from `Command` is passed to the operation `all_elements_do` to be applied to all elements of the collection.

```
interface Command {  
    boolean do_on (in any element);  
};
```

The Comparator Interface

An instance of a user defined interface derived from `Comparator` is passed to the operation `sort` as sorting criteria.

```
interface Comparator {  
    long compare (in any element1, in any element2);  
};
```

The `compare` operation of the user's comparator (interface derived from `Comparator`) must return a result according to the following rules:

```
>0    if (element1 > element2)  
0      if (element1 = element2)  
<0    if (element1 < element2)
```

Appendix A *OMG Object Query Service*

A.1 *Object Query Service Differences*

Identification and Justification of Differences

The relationship between the Object Collection Service (OCS) and the Object Query Service (OQS) is two-fold. The Object Query Service uses collections as *query result* and as scope of query evaluation.

The `get_result` operation of `CosQuery::Query` for example and the `evaluate` operation of `CosQuery::QueryEvaluator` may return a collection as result or may return an iterator to the query result.

There may be a `QueryEvaluator` implementation that takes a collection instance passed as input parameter to evaluate a query on this collection which specifies the scope of evaluation. The query evaluator implementation relies on the `Collection` interface and the generic `Iterator` being supported by the collection passed.

A `CosQuery::QueryableCollection` is a special case of query evaluator which allows a collection to serve directly as the scope to which a query may be applied. As `QueryableCollection` is derived from `Collection` a respective instance can serve to collect a query result to which further query evaluation is applied.

Both usages of collections - as query result and as scope of evaluation - rely on the fact that a minimum collection interface representing a generic aggregation capability is supported as a common root for all collections. Further, they rely on a generic iterator that can be used on collections independent of their type.

Summarizing, Object Query Service essentially depends on a generic collection service matching some minimal requirements. As Object Query Service was defined when there was not yet any Object Collection Service specification available a generic collection service was defined as part of the Query Service specification.

The `CosQueryCollection` module defines three interfaces:

- `CollectionFactory`: provides a generic creation capability
- `Collection`: defines a generic aggregation capability
- `Iterator`: offers a minimal interface to traverse a collection.

Those interfaces specify the minimal requirements of OQS to a generic collection service. The following discusses whether it is possible to replace `CosQueryCollection` module by respective interfaces in the `CosCollection` module as defined in this specification. Differences are identified and justified.

In anticipation of the details given in the next paragraph we can summarize:

- The `CosCollection::Collection` top level collection interface matches the `CosQueryCollection::Collection` interface except for minor differences. Collections as defined in the `CosCollection` module can be used with Query Service.
- The `CosCollection::Collection` top level collection interface proposes an operation which one may consider as an overlap with the Object Query Service function. The operation `all_elements_do` which can be considered a special case of query evaluation.
- The `CosCollection::Iterator` top level iterator interface is consistent with `CosQueryCollection::Iterator` interface in the sense that operations defined in `CosQueryCollection::Iterator` are supported in `CosCollection::Iterator`. In addition a managed iterator semantics is defined which is reflected in the specified side effects on iterators for modifying collection operations. This differs from the iterator semantics defined in the Object Query Service specification but is considered a requirement in a distributed environment.
- There are a number of operations in the `CosCollection::Iterator` interface you do not find in the `CosQueryCollection::Iterator` interface. They are defined in the `CosCollection::Iterator` interface to provide support for performing distributed processing of very large collections and to support the generic programming model as introduced with ANSI STL to the C++ world.
- The restricted access collections which are part of this proposal do not inherit from the top level `CosCollection::Collection` interface. They cannot be used with Object Query Service as they are. But this is in the inherent nature of the restricted access semantics of these collections and is not considered to be a problem. Nevertheless, the interfaces of the restricted access collections allow combining them with the collections of the combined property collections hierarchy via multiple inheritance to enable usage of restricted access collections within the Object Query Service. In doing so, the restricted access collections lose the guarantee for restricted access, but only support interfaces offering the commonly used operation names for convenience.
- The `CosQueryCollection::CollectionFactory` defines the exact same interface as `CosCollection::CollectionFactory`.

Replacing the interfaces defined in the Object Query Service `CosQuery::Collection` module by the respective interface defined in this specification, the Object Collection Service enables the following inheritance relationship:

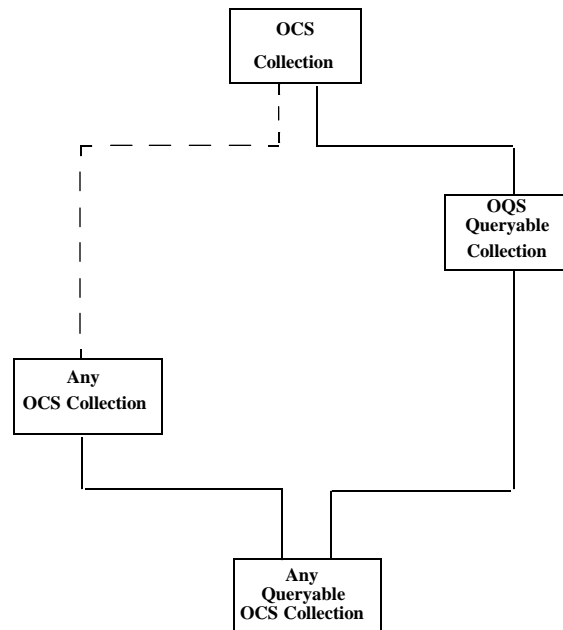


Figure 17-4 Inheritance Relationships

A detailed comparison of the interfaces is given in the following sections and is outlined along the `CosQueryCollection` module definitions.

CosQueryCollection Module Detailed Comparison

Exception Definitions

The following mapping of exceptions holds true:

- `CosQueryCollection::ElementInvalid` maps to `CosCollection::ElementInvalid`
- `CosQueryCollection::IteratorInvalid` maps to `CosCollection::IteratorInvalid` (with `IteratorInvalidReason not_for_collection`)
- `CosQueryCollection::PositionInvalid` maps to `CosCollection::IteratorInvalid` (with `IteratorInvalidReason is_invalid`) and `CosCollection::IteratorInBetween`

Type Definitions

There are a number of type definitions in the `CosQueryCollection` module for the mapping of SQL data types and for defining the type `Record`. These types are Object Query Service specific; therefore, they are not part of the Object Collection Service defined in this specification. Object Query Service may move these definitions to the `CosQuery` module.

CollectionFactory Interface

The `CosQueryCollection::CollectionFactory` interface defines the same interface as `CosCollection::CollectionFactory` and with it the same generic creation capability.

While the generic create operations of `CosQueryCollection::CollectionFactory` do not raise any exceptions, the respective operation in the `CosCollection::CollectionFactory` raises exception “`ParameterInvalid`.”

Collection Interface

The `CosQueryCollection::Collection` interface defines a basic collection interface, without restricting specializations to any particular type such as equality collections or ordered collections.

Collection Element Type

The element type of Object Query Service collections is a CORBA any to meet the general requirement that collections have to be able to collect elements of arbitrary type. The same holds true for the proposed Object Collection Service defined in this specification.

Using the CORBA any as element type implies the loss of compile time type checking. The Object Collection Service as defined here-in considers support for run-time type checking as important; therefore, it offers respective support. In the interface `Collection` this is reflected by introducing a read-only attribute “`element_type`” of type `TypeCode` which enables a client to inquiry the element type expected.

This differs from Object Query Service collections which do not define any type checking specific support.

Collection Attributes

The following attribute is defined in the OQS Collection interface:

cardinality

This read-only attribute maps to the operation `number_of_elements()` in `CosCollection::Collection`. This is semantically equivalent. The name of the operation was chosen consistently with the overall naming scheme of the Collection Service.

Collection Operations

The following operations are defined in the Object Query Service Collection interface.

void add_element (in any element) raises (ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

`boolean add_element(in any element) raises (ElementInvalid)`

void add_all_elements (in Collection elements) raises (ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void add_all_from (in Collection collector) raises (ElementInvalid).

void insert_element_at (in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

boolean add_element_set_iterator(in any element, in Iterator where) raises (IteratorInvalid, ElementInvalid).

void replace_element_at (in any element, in Iterator where) raises (IteratorInvalid, PositionInvalid, ElementInvalid);

This operations maps to

void replace_element_at (in Iterator where, in any element) raises (IteratorInvalid, IteratorInBetween,ElementInvalid).

void remove_element_at (in Iterator where) raises (IteratorInvalid, PositionInvalid)

This operation maps - except for side effects on iterators due to managed iterator semantics - to

void remove_element_at (in Iterator where) raises (IteratorInvalid, IteratorInBetween).

void remove_all_elements ()

This operation maps - except for side effects on iterators due to managed iterator semantics - to

unsigned long remove_all ().

any `retrieve_element_at` (in `Iterator` where) raises (`IteratorInvalid`, `PositionInvalid`)

This operation maps to

boolean `retrieve_element_at` (in `Iterator` where, out any element) raises (`IteratorInvalid`, `IteratorInBetween`).

`Iterator create_iterator ()`

This operation maps to

`Iterator create_iterator` (in boolean `read_only`).

The parameter “`read_only`” parameter is used to support `const` iterators. This is introduced to support the iterator centric ANSI STL like programming model.

Where different operation names are used in the Object Collection Service defined here-in this is done to maintain consistency with the Collection Service overall naming scheme.

Side effects to iterators specified differ from those specified in the Query Service collection module as the Object Collection Service defined here-in specifies a managed iterator model which we consider necessary in a distributed environment. For more details in the managed iterator semantics see chapter “Iterator Interfaces.”

The top-level `CosCollection::Collection` interface proposes all the methods defined in `CosQueryCollection::Collection`. There are some few additional operations defined in `CosCollection::Collection`:

boolean `is_empty()`

This operation is provided as there are collection operations with the precondition that the collection must not be empty. To avoid an exception, the user should have the capability to test whether the collection is empty.

void `destroy()`

This operation is defined for destroying a collection instance without having to support the complete `LifeCycleObject` interface.

void `all_elements_do`(in `Command` `command`)

This operation is added for convenience; however, it seems to be an overlap with OQS functionality. This frequently used trivial query should be part of the collection service itself. A typical usage of this operation may be, for example, iterating over the collection to print all element values. Note that the `Command` functionality is very restricted to enable an efficient implementation. That is, the command is not allowed to change the positioning property of the element applied to and must not remove the element.

Iterator Interface

The `CosQueryCollection::Iterator` corresponds to `CosCollection::Iterator`. `CosCollection::Iterator` is supported for all collection interfaces of the Object Collection Service derived from `Collection`. The Object Collection Service iterator interfaces defined in this specification are designed to support an iterator centric and generic programming model as introduced with ANSI STL. This implies very powerful iterators which go far beyond simple pointing devices as one needs to be able to retrieve, add, remove elements from/to a collection via an iterator. In addition iterator interfaces are enriched with bulk and combined operations to enable an efficient processing of collections in distributed scenarios. Subsequently, the `CosCollection::Iterator` is much more powerful than the `CosQueryCollection::Iterator`.

Iterator Operations

The following operations are defined in the `CosQueryCollection::Iterator` interface:

- any `next ()` raises (`IteratorInvalid`, `PositionInvalid`)

This operation maps to

`boolean retrieve_element_set_to_next (out any element)` raises (`IteratorInvalid`, `IteratorInBetween`)

- `void reset ()`

This operation maps to

`boolean set_to_first_element()` of the Object Collection Service Iterator interface.

- `boolean more ()`

This operation maps to

`boolean is_valid() && ! is_inbetween()`

Due to the support for iterator centric and generic programming there are number of additional operations in the `CosCollection::Iterator` interface:

- `set_to_next_element`, `set_to_next_nth_element`
- `retrieve_element`, `retrieve_next_n_elements`,
`not_equal_retrieve_element_set_to_next`
- `remove_element`, `remove_element_set_to_next`, `remove_next_n_elements`,
`not_equal_remove_element_set_to_next`
- `replace_element`, `replace_element_set_to_next`, `replace_next_n_elements`,
`not_equal_replace_element_set_to_next`
- `add_element_set_iterator`, `add_n_elements_set_iterator`
- `invalidate`
- `is_in_between`, `is_for`, `is_const`, `is_equal`
- `clone`, `assign`, `destroy`

Most of the operations can be implemented as combinations of other basic iterator operations so that the burden put on Object Query Service providers who implement such an interface should not be too high.

A.2 *Other OMG Object Services Defining Collections*

There are several object services that define collections, that is Naming Service, Property Service, and the OMG RFC "System Management: Common Management Facility, Volume 1" submission, for example.

These services define very application specific collections. The Naming Service for example defines the interface `NamingContext` or the Property Service an interface `PropertySet`. Both are very application specific collections and may be implemented using the Object Collection Service probably wrapping an appropriate Object Collection Service collection rather than specializing one of those collection interfaces.

The collections defined in the System Management RFC form a generic collection service. But the service defines collection members that need to maintain back references to collections in which they are contained to avoid dangling references in collections. This was considered as inappropriate heavyweight for a general object collection service. The collections in the System Management RFC may use Object Collection Service collections for their implementation up to some extent even reuse interfaces.

A.3 *OMG Persistent Object Services*

Collections as persistent objects in the sense defined by the Persistent Object Service

- may support the `CosPersistencePO::PO` interface. This interface enables a client being aware of the persistent state to explicitly control the PO's relationship with its persistent data (connect/disconnect/store/restore)
- may support the `CosPersistence::SD` interface which allows objects to synchronize their transient and persistent data
- have to support one of protocols used to get persistent data in and out of an object, like DA, ODMG, or DDO.

Support for these interfaces does not effect the collection interface.

Persistent *queryable* collections may request index support for collections. "Indexing of collections" enables to exploit underlying indices for efficient query evaluation. We do not consider "indexed collections" as part of the Object Collection Service but think that indexing support can be achieved via composing collections defined in the Object Collection Service proposed.

A.4 *OMG Object Concurrency Service*

Any implementation of the Object Collection Service probably will have to implement concurrency support. But we did not define any explicit concurrency support in the collection interfaces as part of the Object Collection Service because we consider that

as an implementation issue that can be solved by specialization. This also would allow to reuse the respective interfaces of the Object Concurrency Service rather than introducing a collection specific support for concurrency.

Appendix B *Relationship to Other Relevant Standards*

B.1 *ANSI Standard Template Library*

The ISO/ANSI C++ standard, as defined by ANSI X3J16 and OSI WG21, contains three sections defining the Containers library, the Iterators library and the Algorithms library, which form the main part of the **Standard Template Library**. Each section describes in detail the class structure, mandatory methods and performance requirements.

Containers

The standard describes two kinds of container template classes, sequence containers and so called associative containers. There is no inheritance structure relating the container classes.

Sequence containers organize the elements of a collection in a *strictly linear* arrangement. The following sequence containers are defined

- **vector**: Is a generalization of the concept of an ordinary C++ array the size of which can be dynamically changed. It's an indexed data structure, which allows fast, that is, constant time random access to its elements. Insertion and deletion of an element at the end of a vector can be done in constant time. Insertion and deletion of an element in the middle of the data structure may take linear time.
- **deque**: Like a vector it is an indexed structure of varying size, allowing fast, that is, constant time random access to its elements. In addition to what a vector offers a deque also offers constant time insertion and deletion of an element at the beginning.
- **list**: Is a sequence of varying size. Insertion and deletion of an element at any position can be done in constant time. But only linear-time access to an element at an arbitrary position is offered.

Associative containers provide the capability for fast, $O(\log n)$, retrieval of elements from the collections by "contents", that is, key value. The following associative containers are provided:

- **set**: Is a collection of unique elements which supports fast access, $O(\log n)$, to elements by element value.
- **multiset**: Allows multiple occurrences of the same element and supports fast access, $O(\log n)$, to elements by value.
- **map**: Is a collection of (key, value) pairs which supports unique keys. It is an indexed data structure which offers fast, $O(\log n)$, access to values by key.
- **multimap**: Is a collection of (key, value) pairs which allows multiple occurrences of the same key.

Container adapters are the well known containers with restricted access, that is:

- **stack**

- queue
- priority_queue

As roughly sketched ANSI STL specifies performance requirements for container operations. Those enforce up to some extent the kind of implementation. If you look at the performance requirements for vector, deque and list they correspond to array and list like implementations.

This differs from what the here-in discussed Object Collection Service proposes. The collection classes vector, deque, and list all map to the same interface Sequence. The different performance profiles are delivered via the implementation choice.

Algorithms

Different from other container libraries ANSI STL containers offer a very limited set of operations at the containers themselves. Instead, all higher level operations like union, find, sort, and so on are offered as so called generic algorithms. A generic algorithm is a global template function that operates on all containers - supporting the appropriate type of iterator. There are approximately 50 algorithms offered in ANSI STL.

There are:

- non-mutating sequence algorithms
- mutating sequence algorithms
- sorting and related algorithms
- generalized numeric algorithms

The basic concept here is the separation of data structures and algorithms. Instead of implementing an algorithm for each container in the library you provide a generic one operating on all containers.

If one implements a new container and ensures that an appropriate iterator type is supported one gets the respective algorithms “for free”. One may also implement new generic algorithms working on iterators only which will apply to all containers supporting the iterator type.

In addition, because the algorithms are coded as C++ global template functions, reduction of library and executable size is achieved (selective binding).

Iterators

The key concept in ANSI STL that enables flexibility of STL are Iterator classes. Iterator classes in ANSI STL are C++ pointer abstractions. They allow iteration over the elements of a container.

Their design ensures, that all template algorithms work not only on containers in the library but also on built-in C++ data type array. Algorithms work on iterators rather than on the containers themselves. An algorithm does not even “know” whether it is working with an ordinary C++ pointer or an iterator created for a container of the library.

There are:

- input iterator, output iterator
- forward iterator
- bidirectional iterator
- random access iterator
- const, reverse, insert iterators

Consideration on choice

The collection class concept as defined by the ANSI standard is designed for optimal, local use within programs written in C++. In some sense they are extensions of the language and heavily exploit C++ language features. No considerations, of course, are given to distribution of objects or language neutrality.

Some of the advantages clearly visible in a local C++ environment cannot be carried over into a distributed and language neutral environment. Some of them are even counterproductive.

In summary, the following list of issues are the reason why the ANSI collection class standard has not been considered as a basis for this proposal:

- Aiming with its design at high performance and small code size of C++ applications ANSI STL seems to have avoided inheritance and virtual functions. As no inheritance is defined, polymorphic use of the defined collection classes is not possible.
- The ANSI STL programming model of generic programming is very C++ specific one. ANSI STL containers, iterators, and algorithms are designed as C++ language extension. Containers are smooth extensions of the built-in data type array and iterators are smooth extensions of ordinary C++ pointers. Container in the library are processed by generic algorithms via iterators in the same way as C++ arrays via ordinary pointers. Rather than subclassing and adding operations to a container one extends a container by writing a new generic algorithm. This is a programming model just introduced to the C++ world with ANSI STL and for sure not the programming model Smalltalk programmers are used to.
- As a consequence of the separation of data structures and algorithms containers in ANSI STL up to some extent expose implementation. As an example consider the two sequential containers `list` and `vector`. The algorithms `sort` and `merge` are methods of the `list` container. `vector` on the other hand can support efficient random access and therefore use the generic

algorithms sort and merge. Subsequently you do not find them as methods in the vector interface. This requires rework of clients when server implementations changes from list to vector or deque because of changing access patterns.

- The IDL concept has no notion of global (template) functions. The only conceivable way to organize the algorithms is by collecting them in artificial algorithm object(s). The selective binding advantage is lost in a CORBA environment and careful placement of the algorithm object(s) near the collection must be exercised.
- In the ANSI STL approach the reliance on generic programming as algorithms is substantial. We believe that this concept is not scalable. It is difficult to imagine a generic sort in a CORBA environment is effective without the knowledge of underlying data structures. Each access to a container has to go via an iterator mediated somehow by the underlying request broker, which is not a satisfactory situation. Object Collection Services will be used in a wide variety of environments, ranging from simple telephone lists up to complex large stores using multiple indices, exhibiting persistent behavior and concurrently accessed via Object Query Service. We do not believe that generic algorithms scale up in such environments.

B.1.1 ODMG-93

Release 1.1 of the ODMG specification defines a set of collection templates and an iterator template class.

An abstract base class `Collection<T>` is defined from which all concrete collections classes are derived. The concrete collection classes supported are `Set<T>`, `Bag<T>`, `List<T>`, `Varray<T>`. In addition an Iterator class `Iterator<T>` is defined for iteration over the elements of the collection.

`Set` and `Bag` are unordered collections and `Bag` allows multiples. `List` is an ordered collection that allows multiples. The `Varray<T>` is a one dimensional array of varying length.

`Collection<T>` offers the test `empty()` and allows to ask for the current number of elements, `cardinality()`. Further the tests `is_ordered()` and `allows_duplicates()` are offered. There is a test on whether an element is contained in a given collection. Operations for insertion, `insert_element()`, and removal, `remove_element()` are provided. Last not least there is a `remove_all()` operation.

Each of the derived classes provides an `operator==` and an `operator!=` and an operation `create_iterator()`.

A `Set<T>` is derived from `Collection<T>` and offers in addition operations `is_subset_of()`, `is_proper_subset_of()`, `is_superset_of()`, or `proper_superset_of()` a suite of set-theoretical operations to form the union, difference, intersection of two sets.

A `Bag<T>` offers the same interface as `Set<T>` but allows multiples.

A `List<T>` offers specific operations to retrieve or remove the first respectively last element in the list or to insert an element as first respectively last element. Retrieving, removing, and replacing an element at a given position is supported. Inserting an element before or after a given position is possible.

`Varray<T>` exposes the characteristics of a one dimensional array of varying length. An array can be explicitly re-sized. The `operator[]` is supported. The operations to find, remove, retrieve, and replace an element at a given position are supported.

An instance `Iterator<T>` is created to iterate over a given collection. The `operator=` and `operator ==` are defined. There is a `reset()` operation moving an iterator to the beginning of the collection. There is an operation `advance()` and overloaded the `operator++` to move the iterator to the next element. Retrieving and replacing the element currently “pointed to” is possible. A check on whether iteration is not yet finished is offered, `not_done()`. For convenience in iteration there is an operation `next()`, combining “check end of iteration, retrieval of an element, and moving to the next element”.

ODMG-93 structure is very similar to the proposed Object Collections Service. ODMG-93 `Set <T>` and `Bag<T>` correspond very well to `Set` and `Bag` as defined herein. `List<T>` maps one-to-one to an `EqualitySequence`. A `Varray<T>` maps to an `EqualitySequence` too. That the interfaces `List<T>` and `Varray <T>` map to the same interface in the Object Collection Service proposed reflects that `List<T>` and `Varray<T>` somehow expose the underlying kind of implementation structure assumed - namely a list like structure respectively a table like structure. In the Object Collection Service proposed the different kinds of implementation of a sequence like interface are not reflected in the interface but only in the delivered performance profile. This is the reason why `List<T>` and `Varray<T>` map to the same interface `EqualitySequence`. The `Iterator` interface maps to the top level `Iterator` interface of the iterator hierarchy of the Object Collection Service.

In summary the Object Collection Service proposed is a superset of the ODMG-93 proposed collections and iterators.

Appendix C References

C.1 List of References

OMG, *CORBAservices: Common Object Services Specification*, Volume 1, March 1996.